A STUDY OF TRANSIENT BOTTLENECKS: UNDERSTANDING AND REDUCING LATENCY LONG-TAIL PROBLEM IN N-TIER WEB APPLICATIONS

A Thesis
Presented to
The Academic Faculty

by

Qingyang Wang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2014

A STUDY OF TRANSIENT BOTTLENECKS: UNDERSTANDING AND REDUCING LATENCY LONG-TAIL PROBLEM IN N-TIER WEB APPLICATIONS

Approved by:

Professor Dr. Calton Pu, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Dr. Ling Liu
School of Computer Science
Georgia Institute of Technology

Professor Dr. Karsten Schwan
School of Computer Science
Georgia Institute of Technology

Professor Dr. Mustaque Ahamad
School of Computer Science
Georgia Institute of Technology

Professor Dr. Shamkant B. Navathe
School of Computer Science
Georgia Institute of Technology

Professor Dr. Doug B. Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: June 18, 2014

To my parents

ACKNOWLEDGEMENTS

The journey to obtain a Ph.D. is a precious and unique experience of my personal growth. I am truly thankful to everyone who has accompanied and supported me through the peaks and valleys of this journey. I apologize in advance if I miss anyone below.

I am very fortunate and grateful to work with my advisor, Dr. Calton Pu. First of all he gave me the opportunity to start my Ph.D. journey in one of the best computer science programs in the world. During my Ph.D. journey his constant guidance and encouragement have been the most important driving force for my growth. He pushed me to exceed my own limit and continuously expand my comfort zone in every aspect of my research. For example, every time I'm satisfied with my little progress in research he reminds me that "you can only learn more when you try hard to jump out of your comfortable zone". I understand that it is his such attitude that makes him a great researcher. Apart from professional advices, Calton has been amazingly generous with attention on other topics, including presentation skills, strategies for job search, and life wisdom based on his experience. I have learned an innumerable amount from him, and I am deeply indebted to him for his constant and unconditional support throughout my years at Georgia Tech.

I am also deeply thankful to Mr. Yasuhiko Kanemasa, my long-term research collaborator and friend from Fujitsu labs. Yasuhiko and I came to Georgia Tech at the same year 2007. He was a visiting scholar and I was the first year Ph.D. student. We soon became very good friends due to his good personality and our common interest. Since then the more I know him, the more I'm impressed by his focus, discipline and strive for perfection in his work. I'm lucky to officially collaborate with him in research starting from 2009. Every time technical difficulties impeded my progress, he guided me past them with his insightful comments and suggestions. I'm grateful for all his guidance and everything I have learned from him. It was amazing to have such great people accompany with me along the harsh Ph.D. journey and I appreciate it deeply.

I would also like to thank members of my dissertation committee - Dr. Ling Liu, Dr. Karsten Schwan, Dr. Mustaque Ahamad, Dr. Shamkant Navathe, and Dr. Doug Blough - for reading and commenting on my dissertation. I appreciate all the challenging questions before and during my defense that led to an improved dissertation.

To my labmates in ELBA project Junhee Park, Jack Li, Simon Malkowski, Deepal Jayasinghe, Pengcheng Xiong, Chien'An Lai, Tao Zhu, Chien-An Cho, Aibek Musaev, and Gueyoung Jung, you have supported me all along the way. We have been always sharing both the good and the bad times. Without your company, the hours in the lab would have gone by much slower and I would have enjoyed it a lot less. Especially Junhee's high skills of dealing with various unexpected crises not only have made all of us laugh loudly but also enriched everyone's life experience.

Also at Georgia tech, I'm fortunate to meet many great friends who have peppered my Ph.D. journey with cherished memories, including Chiemi Amagasa, Bhuvan Bamba, Qinyuan Feng, Chris Grayson, Binh Han, Liting Hu, Danesh Irani, Minsung Jang, Mukil Kesavan, Younggyun Koh, Balaji Palanisamy, Priyanka Tembey, De Wang, Jinpeng Wei, Qinyi Wu, Yuehua Wang, Chengwei Wang, Rui Zhang, and Fang Zheng. My sincerest thank you! It is you who make my life more colorful and cheerful.

Finally, but most significantly, to my family, I am here and able to enjoy my Ph.D. journey because of you and I love you all dearly. To my mother, Chunjiao Zhang, you are my strongest supporter no matter where I am. To my father, Duyi Wang, I always remember your words of asking me being a sincere and warm person. To my sister Huan Wang, I'm always proud of you and your courage to conquer any difficulties on your journey of creating your own business. To my brother-in-law Ming Zhu, thank you for taking care of my sister and also my super cute niece Xiaoxiao. To my girlfriend Lu Liu, thank you for accompanying with me and supporting me through my Ph.D. journey. You make my life much more colorful and meaningful.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

An essential requirement of cloud computing or data centers is to simultaneously achieve good performance and high utilization for cost efficiency. High utilization through virtualization and hardware resource sharing is critical for both cloud providers and cloud consumers to reduce management and infrastructure costs (e.g., energy cost, hardware cost) and to increase cost-efficiency. Unfortunately, achieving good performance (e.g., low latency) for web applications at high resource utilization remains an elusive goal. Both practitioners and researchers have experienced the latency long-tail problem in clouds during periods of even moderate utilization (e.g., 50%). In this dissertation, we show that transient bottlenecks are an important contributing factor to the latency long-tail problem. Transient bottlenecks are bottlenecks with a short lifespan on the order of tens of milliseconds. Though short-lived, transient bottleneck can cause a long-tail response time distribution that spans a spectrum of 2 to 3 orders of magnitude, from tens of milliseconds to tens of seconds, due to the queuing effect propagation and amplification caused by complex inter-tier resource dependencies in the system. Transient bottlenecks can arise from a wide range of factors at different system layers. For example, we have identified transient bottlenecks caused by CPU dynamic voltage and frequency scaling (DVFS) control at the CPU architecture layer, Java garbage collection (GC) at the system software layer, and virtual machine (VM) consolidation at the application layer. These factors interact with naturally bursty workloads from clients, often leading to transient bottlenecks that cause overall performance degradation even if all the system resources are far from being saturated (e.g., less than 50%). By combining fine-grained monitoring tools and a sophisticated analytical method to generate and analyze monitoring data, we are able to detect and study transient bottlenecks in a systematic way.

CHAPTER I

INTRODUCTION

Wide response time fluctuations (latency long tail problem) of large scale distributed applications at even moderate system utilization levels (e.g., 50%) have been reported both in industry [32] and academia [47, 49, 78, 84]. Occasionally and without warning, some requests that usually return within a few milliseconds would take several seconds. These very long response time (VLRT) requests are difficult to study for two major reasons. First, the VLRT requests only take milliseconds when running by themselves, so the problem is not with the VLRT requests, but emerges from the interactions among system components. Second, the statistical average behavior of system components (e.g., average CPU utilization over typical measurement intervals such as minutes) shows all system components to be far from saturation.

My dissertation research shows transient bottlenecks being an important contributing factor to the latency long tail problem. Transient bottlenecks are bottlenecks with a short lifespan on the order of tens of milliseconds. Though short-lived, transient bottlenecks can cause queue overflows that propagate through an n-tier system, resulting in dropped messages and VLRT requests due to timeout and retransmissions.

Transient bottlenecks can arise from a wide range of factors at different system layers. For example, we have identified transient bottlenecks caused by CPU dynamic voltage and frequency scaling (DVFS) control at the architecture layer, soft resource allocation (e.g., number of threads or database connections) at the software layer, virtual machine (VM) colocation at the middleware layer, and transaction scheduling at the application layer. These factors interact with naturally bursty workloads from clients, often leading to transient bottlenecks that cause overall performance degradation even if all the system resources are far from being saturated (e.g., less than 60%).

The study of transient bottlenecks has been hampered by several reasons. First, many

transient bottlenecks are short-lived (on the order of tens of milliseconds). From Sampling Theory, these transient bottlenecks would not be reliably detectable by typical monitoring tools that sample at time intervals measured in seconds or minutes. These monitoring tools incur high overhead at sub-second sampling intervals (about 6% CPU utilization overhead at 100ms interval and 12% at 20ms interval using Dstat [1]). Second, although our understanding of transient bottlenecks has been limited, practical solutions to bypass the VLRT request problem caused by transient bottlenecks have been described [32]. For example, applications with read-only semantics (e.g., web search) can use duplicate requests sent to independent servers and reduce perceived response time by choosing the earliest answer. These bypass techniques are effective in specific domains, contributing to an increasingly acute need to improve our understanding of the general causes (transient bottlenecks) for the VLRT requests. On the practical side, our lack of a detailed understanding of transient bottlenecks is consistent with the low average overall data center utilization [74] at around 18%, which is a more general way to avoid VLRT requests. The current situation shows that transient bottlenecks certainly merit further investigation and better understanding, both as an intellectual challenge and their potential practical impact (e.g., to increase the overall utilization and return on investment in data centers).

By combining fine-grained monitoring tools (a combination of microsecond resolution message timestamping and millisecond system resource sampling) and sophisticated analytical methods to generate and analyze monitoring data, we are able to describe, understand, and remedy transient bottlenecks in a systematic way.

1.1 Dissertation Statement and Contributions

My dissertation statement is formulated as follows:

Thesis Statement: *Systematic and empirical understanding of transient bottlenecks (defined as a transient CPU saturation period on the order of tens of milliseconds) through fine-grained monitoring can contribute to effectively reducing the latency long tail problem in n-tier web applications.*

To support my dissertation statement, we make the following concrete contributions:

- **Our first contribution is a set of micro-level event analyses of fine-grained experimental data that link transient bottlenecks with varied causes to the very long response time (VLRT) requests.** Our micro-level event analyses includes five steps: (1) VLRT requests are detected in an n-tier system with moderate utilization; (2) at the same time, long request queues are formed in the Apache overflowing TCP buffer, causing dropped packets and retransmission after three seconds (VLRT requests); (3) the long queues in Apache are formed because the downstream server (Tomcat) became saturated and the corresponding queue soon filled up, causing Tomcat to block new requests from Apache.(4) long queues in Tomcat servers are created by transient bottlenecks, in which the server CPU becomes saturated for a very short period of time. (5) transient bottlenecks are associate with a specific root cause. Through extensive measurements of an n-tier application benchmark (RUBBoS [8]), we have found that transient bottlenecks can arise from varied causes in different system layers, including CPU dynamic voltage and frequency scaling (DVFS) control at the architecture layer, Java garbage collection (GC) at the system software layer, and virtual machine (VM) consolidation at the application layer.

- **Our second contribution is a novel transient bottleneck detection method, which detects transient bottlenecks with varied causes and enable our micro-level event analyses (the first contribution) through the fine-grained measurement data.** Our method uses passive network packet tracing, which monitors the arrival and departure timestamps of each request of each server in the system at microsecond granularity [80]. This data supports the counting of concurrent requests and completed requests of each server at fine time granularity (e.g., 50ms). For sufficiently short time intervals, we can use the server request completion rate as throughput, and concurrent requests as server load, to identify transient performance bottlenecks (Utilization Law [33]) at time granularity as short as 50ms. Since this

method is completely independent of specific resource saturation measurements, transient bottlenecks with varied causes even in different system layers can be detected. In addition, with the precise arrival and departure timestamps of each request of each server, we also have the exact knowledge of the occurrence of VLRT requests and the number of queued requests in each tier (e.g., Tomcat and Apache) at each time window. Such fine-grained measurement data are essential for our micro-level event analyses that link transient bottlenecks to the occurrence of VLRT requests (see the first contribution).

- **The third contribution is a systematic discussion of remedies for VLRT requests caused by transient bottlenecks.** We first discuss specific solutions for two identified causes of transient bottlenecks for VLRT requests, for example, Java GC was streamlined from JVM 1.5 to 1.6 and the transient bottlenecks caused by the Dell BIOS-level DVFS controller can be reduced using workload-sensitive adaptive control. On the other hand, VLRT requests arise from statistical coincidences such as VM consolidation (a kind of noisy neighbor problem) and cannot be easily "fixed" through specific solutions. Using transient bottlenecks, we discuss the limitations of some potential solutions (e.g., making queues deeper through additional soft resource allocations causes bufferbloat) and describe generic remedies to reduce or bypass the queue amplification process (e.g., through the separation of short requests from resource-intensive requests to reduce queuing of short requests).

1.2 Organization of This Dissertation

The rest of this dissertation is organized as follows. Chapter 2 shows concrete experimental evidence of the latency long-tail problem using a standard n-tier web application benchmark (RUBBoS [8]) running at high utilization. We found that the latency long-tail problem is due to some requests, which would normally finish within tens of milliseconds, having very long response times (VLRT). Chapter 3 describes the micro-level event analyses that explicitly link various causes to VLRT requests, based on the fine-grained measurement data collected from different system layers. Chapter 4 shows a generic transient bottleneck

detection method, which is sensitive enough to detect transient bottlenecks at millisecond level and with negligible performance overhead for the runtime application. Chapter 5 discusses some specific and general remedies for reducing or avoiding VLRT requests caused by transient bottlenecks. Chapter 6 summarizes the related work and Chapter 7 concludes the dissertation and discusses future work.

CHAPTER II

A EXPERIMENTAL STUDY OF LATENCY LONG-TAIL PROBLEM

In this chapter, we show concrete experimental evidence of the latency long-tail problem using a standard n-tier web application benchmark (RUBBoS [8]) running at high utilization. We found that except the bursty workload from clients, the latency long-tail problem can be caused by some system environmental conditions (e.g., L2 cache miss, JVM garbage collection, inefficient scheduling policies) that commonly exist in n-tier applications. The impact of these system environmental conditions can largely amplify the end-to-end response time because of the complex resource dependencies in the system. For instance, a 50ms response time increase in the database tier can be amplified to 500ms end-to-end response time increase. Our results show that the latency long-tail problem should be taken into account when designing effective autonomous self-scaling n-tier systems in cloud.

2.1 Introduction

Simultaneously achieving good performance and high resource utilization is an important goal for production cloud environments. High utilization is essential for high return on investment for cloud providers and low sharing cost for cloud users [38]. Good performance is essential for mission-critical applications, e.g., web-facing e-commerce applications with Service Level Agreement (SLA) guarantees such as bounded response time. Unfortunately, simultaneously achieving both objectives for applications that are *not* embarrassingly parallel has remained an elusive goal. Consequently, both practitioners and researchers have encountered large response time fluctuations (latency long-tail problem) in clouds during periods of high utilization. A practical consequence of this problem is that enterprise cloud environments have been reported to have disappointingly low average utilization (e.g., 18% in [74]).

Using extensive measurements of an n-tier benchmark (RUBBoS [8]), we show concrete experimental evidence of the latency long-tail problem. The latency long-tail, ranging from

tens of milliseconds up to tens of seconds, appears when workloads become bursty [55], as expected of web-facing applications. The discovery of the latency long-tail problem is important as it will have significant impact on the autonomous performance prediction and tuning of n-tier application performance, even for moderately bursty workloads. Specifically, a distinctly bi-modal distribution with two modes (that span a spectrum of 2 to 3 orders of magnitude) can cause significant distortions on traditional statistical analyses and models of performance that assume uni-modal distributions.

One of the interesting facts that made this research challenging is that the long queries (that last several seconds) are not inherently complex in their nature, i.e., they are normal queries that would finish within tens of milliseconds when run by themselves. Under a specific (and not-so-rare) set of system environmental conditions, these queries take several seconds. The detailed analysis to reveal these system environmental conditions in an n-tier system is non-trivial considering that classical performance analysis techniques that assume uni-modal distributions are inapplicable. Our approach recorded both application level and system level metrics (e.g., response time, throughput, CPU, and disk I/O) of each tier in an n-tier system at fine-grained time granularity (e.g., 100ms). Then we analyzed the relationship of these metrics among each tier to identify the often shifting and sometimes mutually dependent bottlenecks. The complexity of this phenomenon is illustrated by a sensitivity study of soft resource allocation (e.g., number of threads in the web and application servers and DB connection pool) on system performance and resource utilization.

The first contribution of the chapter is an experimental illustration of the latency long-tail problem of systems under high resource utilization conditions using the n-tier RUBBoS benchmark. Due to the large fluctuations, the average system response time is not representative of the actual system performance. For instance, when the system is under a moderately bursty workload and the average utilization of the bottleneck resource (e.g., MySQL CPU) is around 90%, the end-to-end response time shows a distinctly bi-modal distribution (Section 2.2.2).

The second contribution of the chapter is a detailed analysis of several system environmental conditions that cause the latency long-tail problem. For instance, some transient

events (e.g., CPU overhead caused by L2 cache miss or Java GC, see Section 2.4.1) in the tier under high resource utilization conditions significantly impact the response time fluctuations of the tier. Then the in-tier response time fluctuations is amplified to the end-to end response time due to the complex resource dependencies across tiers in the system (Section 2.4.2). We also found that the operating system (OS) level "best" scheduling policy in each individual tier of an n-tier system may not achieve the best overall application level response time (Section 2.4.3).

The rest of the chapter is organized as follows. Section 2.2 shows the latency long-tail problem using a concrete example. Section 2.3 illustrates our fine-grained monitoring analysis. Section 2.4 shows some system environmental conditions for the latency long-tail problem. Section 2.5 summarizes the related work and Section 2.6 concludes the paper.

2.2 Background and Motivation

2.2.1 Experimental Setup

In our experiments we adopt the RUBBoS n-tier benchmark, based on bulletin board applications such as Slashdot [8]. RUBBoS has been widely used in numerous research efforts due to its real production system significance. RUBBoS can be configured as a three-tier (web server, application server, and database server) or four-tier (addition of clustering middleware such as C-JDBC [26]) system. The workload generator of this benchmark emulates a number of users interacting with the web application. Each user has an average 7-second thinking time between receiving a web page and submitting a new page download request. The workload includes 24 different interactions such as "register user" or "view story". The benchmark includes two kinds of workload modes: browse-only and read/write interaction mixes. We use browse-only workload for our evaluation.

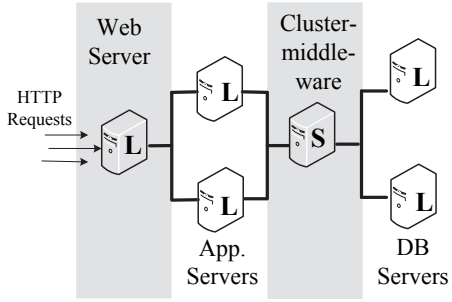
Mi et al. [55] proposed a *bursty workload generator* which takes into account the Slashdot effect, where a web page linked by a popular blog or media site suddenly experiences a huge increase in web traffic [11]. Unlike the original workload generator which generates a request rate that follows a Poisson distribution parameterized by a number of emulated clients, the bursty workload generator generates request rates in two modes: a fast mode with short

Function	Software
Web Server	Apache 2.0.54
Application Server	Apache Tomcat 5.5.17
Cluster middleware	C-JDBC 2.0.2
Database server	MySQL 5.0.51a
Sun JDK	jdk1.6.0_14
Operating system	RHEL Server 5.7 (Tikanga)
System monitor	Sysstat 10.0.02, Collectl 3.5.1
Transaction monitor	Fujitsu SysViz

(a) Software setup

Hardware	Processor # cores	Freq.	L2 Cache	Memory	Disk	Network
Large (L)	2	2.27GHz	2M	2GB	200GB	1Gbps
Medium (M)	1	2.4 GHz	4M	2GB	200GB	1Gbps
Small (S)	1	2.26GHz	512k	1GB	80GB	1Gbps

(b) Hardware node setup

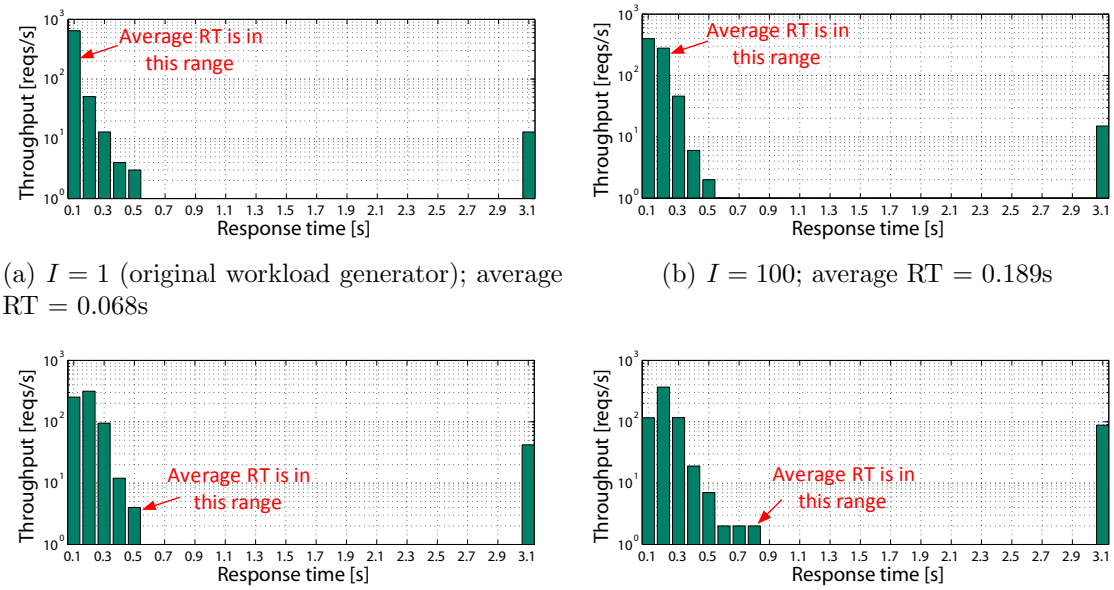


(c) 1L/2L/1S/2L sample topology

Figure 1: Details of the experimental setup.

user think time and a slow mode with long user think time. The fast mode simulates the Slashdot effect where the workload generator generates traffic surges for the system. The bursty workload generator uses one parameter to characterize the intensity of the traffic surges: *index of dispersion*, which is abbreviated as I. The larger the I is, the longer the duration of the traffic surge. In this chapter, we use both the original workload generator (with $I = 1$) and the bursty workload generator (with $I = 100$, 400, and 1000) to evaluate the system performance.

The details of the experimental setup is in Figure 1. We carry out the experiments by allocating a dedicated physical node to each server. A four-digit notation $\#W/\#A/\#C/\#D$ is used to denote the number of web servers, application servers, clustering middleware servers, and database servers. We have three types of hardware nodes: "L", "M", and "S", each of which represents a different level of processing power. Figure 1(c) shows a sample 1L/2L/1S/2L topology. Hardware resource utilization measurements are taken during the runtime period using collectl [4] at different time granularity. We use Fujitsu SysViz [2], a prototype tool developed by Fujitsu laboratories, as a system tracing facility



(a) $I = 1$ (original workload generator); average RT = 0.068s

(b) $I = 100$; average RT = 0.189s

(c) $I = 400$; average RT = 0.439s

(d) $I = 1000$; average RT = 0.776s

Figure 2: End-to-end response time distribution of the system in workload 5200 with different burstiness levels; the average CPU utilization of the bottleneck server is 90% in 10 minutes runtime experiments for all the four cases.

that timestamps all network packets in the monitored system at microsecond granularity. By recording the precise arrival and departure timestamps of each client request for each server, we are able to determine precisely the response time and the number of concurrent requests in each server of the monitored system at fine-grained time windows (e.g., 100ms).

2.2.2 Latency Long-Tail Problem at High Utilization

In this section, we give one example to show that the average of measured performance metrics may not be representative of the actual system performance perceived by clients when the system is under high utilization conditions. The results shown here are based on 10-minute runtime experiments of RUBBoS benchmark running in a four-tier system (see Figure 1(c)) with different burstiness levels of workload.

Figure 2 shows the system response time distribution with four different burstiness levels of workload. The sum of the value of each bar in a subfigure is the total system throughput. We note that in all these four cases, the CPU utilization of the bottleneck server (the CJDBC server) of the system is 90%. This figure shows that the response time distribution in each

of these four cases has a distinctly bi-modal characteristic; while majority of requests from clients finish within a few hundreds of milliseconds, a few percentage finish longer than three seconds. Furthermore, this figure shows the more bursty the workload, the more requests there will be with response time longer than 3 seconds.

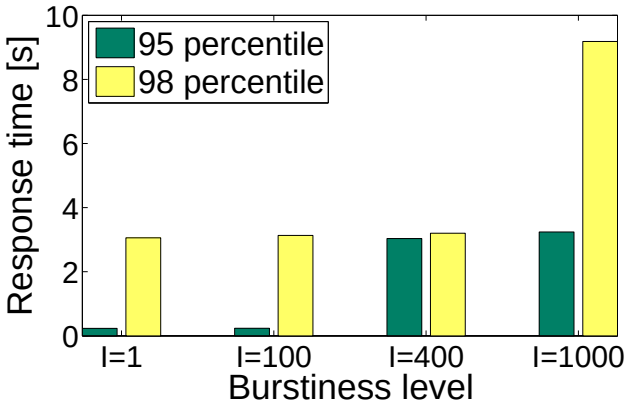
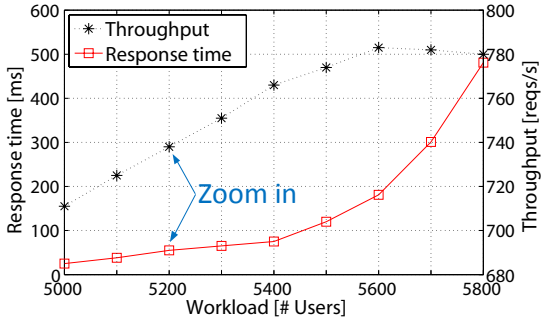


Figure 3: The percentiles of system response time in workload 5200 with different burstiness levels.

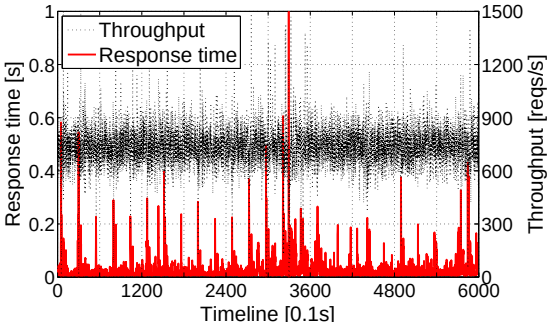
Latency long-tail problem has significant negative impact on the performance of a system requiring strict Service Level Agreement (SLA) guarantees such as bounded response time. Figure 3 shows the 95- and 98-percentiles of the end-to-end response time under different levels of bursty workload. For the original workload ($I = 1$) case and the bursty workload ($I = 100$) case, the 95th percentile is very low (less than 200ms) while the 98th percentile is over 3 seconds. As the burstiness level of workload increases, even the 95-percentile's response time is beyond 3 seconds, and the 98-percentile's for bursty workload ($I = 1000$) case exceeds 9 seconds. Some web-facing applications have strict response time requirement, for example, Google requires clients' requests to be processed within a few hundreds of milliseconds [32]. Thus, latency long-tail may lead to severe SLA violations though the average response time is small.

2.3 Fine-Grained Analysis for the Latency Long-Tail Problem

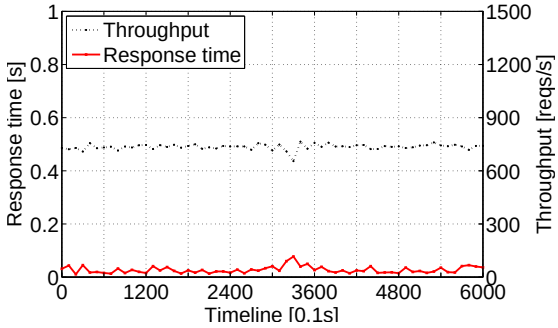
In this section we show the cause of the distinctly bi-modal response time distribution as introduced in the motivation case through fine-grained analysis. The results here are based on the same configuration as shown in the motivation case. We use the original workload



(a) Average End-to-end RT and TP at each workload; (b) and (c) shows the fine-grained measurement value over continuous time windows at workload 5200.



(b) Large fluctuations of RT and TP (average at each 100ms).



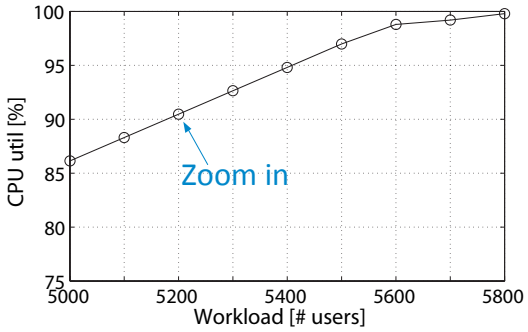
(c) Relatively stable RT and TP (average at each 10s).

Figure 4: Analysis of system response time and throughput using both the macro-level average (see (a)) and micro-level monitoring (see (b) and (c)). (b) shows the large response time fluctuations of the system at workload 5200 while such fluctuations are masked in (b) and (c).

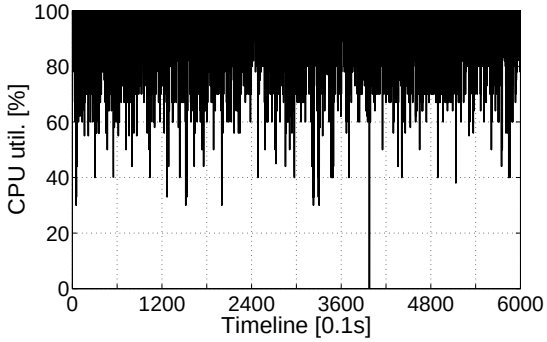
generator ($I = 1$), which is an extension analysis for the case as shown in Figure 2(a).

Figure 4(a) shows the average throughput and response time of the system from workload 5000 to 5800. The response time distribution shown in Figure 2(a) is based on the result of workload 5200, where the average response time is 0.068s and the average CPU utilization of CJDBC server is about 90% (see Figure 5(a)). Next, we zoom in the highly aggregated average of the application/system metrics measured in workload 5200 through fine-grained analysis.

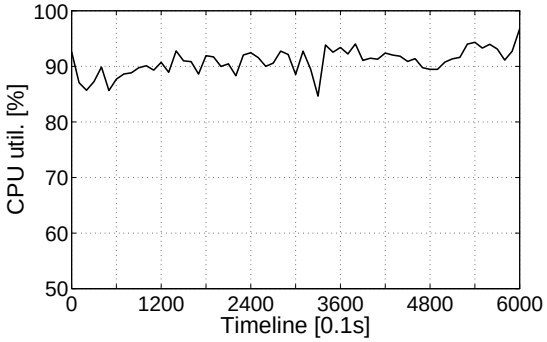
Figure 4(b) and 4(c) show the average system response time and throughput aggregated at 100ms and 10s time granularities respectively. Figure 4(b) shows both the system response time and throughput present large fluctuations while such fluctuations are highly



(a) Bottleneck server CPU usage; (b) and (c) show the "zoom in" results.



(b) Large fluctuations of CPU usage (average at each 100ms).



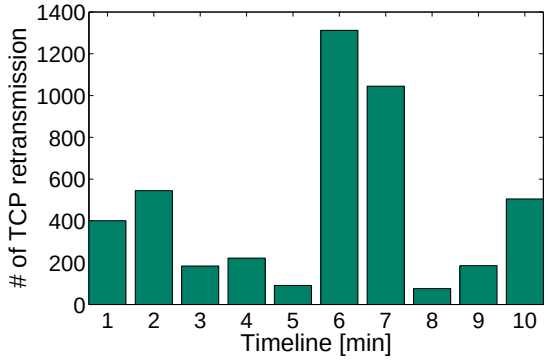
(c) Relatively stable CPU usage (average at each 10s).

Figure 5: Analysis of the bottleneck server CPU utilization using both the macro-level average (see (a)) and micro-level monitoring (see (b) and (c)). (b) shows that CJDBC CPU has frequent transient CPU saturations while such saturations are masked by the average value over a long period.

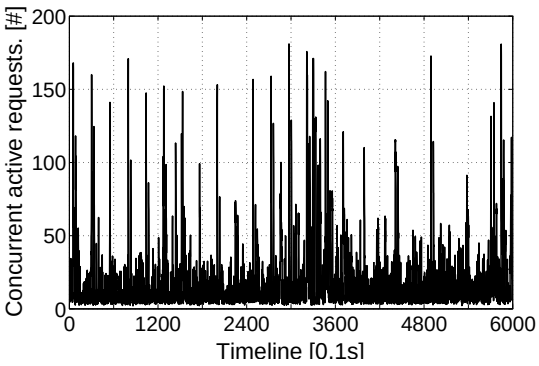
blurred when 10 second time granularity is used (Figure 4(c)). Figure 5(b) and 5(c) show the similar graphs for the CJDBC (the bottleneck server) CPU utilization. Figure 5(b) shows the CJDBC CPU frequently reaches 100% utilization if monitored at 100ms granularity while such CPU saturation disappears if 10s time granularity is used [1].

Figure 6(b) and 6(c) show the number of concurrent requests on the Apache web server aggregated at 100ms and 10s time granularity in workload 5200. Concurrent requests on a server refer to the requests that have arrived, but have not departed from the server; these requests are being processed concurrently by the server due to the multi-threading architecture adopted by most modern internet server designs (e.g., Apache, Tomcat, and

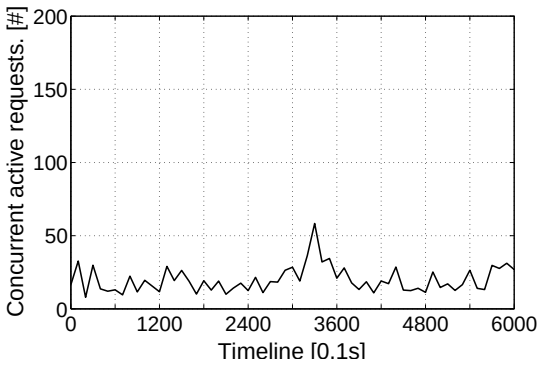
[1] 10 seconds or even longer control interval is frequently used in automatic self-scaling systems [9,50,62,83].



(a) # of TCP retransmission measured in each minute. A TCP retransmission forces a client to backoff at least 3 seconds to resend the dropped request, leading to the very long response time of the request.



(b) Large concurrent request fluctuations in Apache (average in each 100ms).



(c) Relatively stable concurrent requests in Apache (average in each 10s).

Figure 6: Analysis of the requests with very long response time and the number of concurrent requests in Apache web tier. Requests with long response time are due to TCP transmissions (see (a)), which are caused by the high peaks of concurrent requests in Apache (see (b)). The high peaks of concurrent requests in Apache is masked using the average value over a long period (see (c)).

MySQL). We note that the thread pool size we set for the Apache web server in this set of experiments is 50; considering the underlying operating system has a buffer (TCP backlog, the default size is 128) for incoming TCP connection requests from clients, the maximum number of concurrent requests the Apache web server can handle is 178. Once the server reaches the limit, the new incoming requests will be dropped and TCP retransmission happens, which causes the long response time perceived by a client [2]. Figure 6(b) shows

[2]TCP retransmission is transparent to clients; the waiting time is three seconds for the first time and is exponentially increased for the consecutive retransmissions (RFC 2988).

Table 1: Workload (with different burstiness levels) beyond which more than 1% TCP retransmission happens.

Burstiness level	Threshold WL	Bottleneck server CPU util.
I =1	5000	88.1%
I =100	4800	86.3%
I =400	4400	80.4%
I =1000	3800	74.6%

that the concurrent requests, if aggregated at 100ms time granularity, frequently present high peaks which are close to the limit. Such high peaks cause large number of TCP retransmissions as shown in Figure 6(a), which counts the number of TCP retransmissions in every minute during the 10-minute runtime experiment.

2.3.1 Sensitivity Analysis with Different Bursty Workloads

System administrators may want to know under which workload(s) the latency long-tail problem happen. Table 1 shows the minimum workload (with different burstiness levels) under which the system has at least 1% requests that encounter TCP retransmissions. This table shows that both the threshold workload and the corresponding average CPU utilization of the bottleneck server decrease as the burstiness level of workload increases. This further justifies that the evaluation of the latency long-tail problem using fine-grained monitoring is an important and necessary step in autonomic system design.

2.4 System Conditions for the Latency Long-Tail Problem

Understanding the exact causes of latency long-tail problem of an n-tier system under high utilization conditions is important to efficiently utilize the system resources while achieving good performance. In this section we will discuss some system environmental conditions that cause the latency long-tail problem even under the moderately bursty workload from clients. We note that all the experimental results in this section are based on the original RUBBoS browse-only workload ($I = 1$).

2.4.1 Transient Events

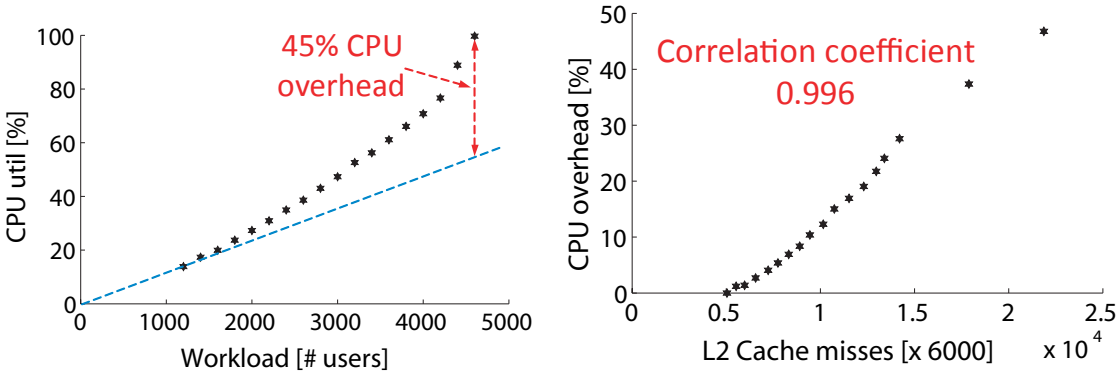
Transient events are events that are pervasive but only happen from time to time in computer systems, such as L2 cache miss, JVM GC, page fault, etc. In this section we will show two types of transient events, L2 cache miss (the last level cache) and JVM GC, that cause significant overhead to the bottleneck resource in the system, especially when the bottleneck tier is in high concurrency of request processing.

2.4.1.1 CPU overhead caused by L2 cache misses

For modern computer architectures, caching effectiveness is one of the key factors for system performance [29, 57]. We found that the number of L2 cache misses of the bottleneck server in an n-tier system increases *nonlinearly* as workload increases, especially when the system is under high utilization conditions. Thus the CPU overhead caused by L2 cache misses significantly impacts the latency long-tail problem of the system.

The hardware configuration of the experiments in this section is 1L/2L/1M (one Apache and two Tomcats on the type "L" machine, and one MySQL on the type "M" machine). Under this configuration, the MySQL server CPU is the bottleneck of the system. We choose the "M" type machine for MySQL as the corresponding Intel $Core^{TM}2$ CPU has two CPU performance counters which allow us to monitor the L2 cache misses during the experiment.

Figure 7(a) shows the MySQL CPU utilization as workload increases from 1200 to 4600 at a 200 increment per step. Ideally the MySQL CPU should increase linearly as workload increases until saturation if there is no CPU overhead. However, this figure clearly shows that the CPU overhead increases nonlinearly as workload increases, especially in high workload range. In order to quantify the CPU overhead and simplify our analysis, we make one assumption here: MySQL has no CPU overhead for request processing from workload 0 to workload 1200 (our starting workload). Under this assumption, we can quantify the CPU overhead for the following increasing workloads by measuring the distance between the actual CPU utilization and the ideal CPU utilization. For instance, under workload 4600, the MySQL CPU overhead reaches 45%.



(a) MySQL CPU overhead

(b) L2 misses vs. CPU overhead

Figure 7: CPU overhead caused by L2 cache misses.

Table 2: Comparison of MySQL CPU utilization and L2 cache misses between DBconn12 and DBconn2 with 1L/2L/1M configuration; higher concurrency leads to more L2 cache misses in the bottleneck tier (MySQL).

WL	DBconn12			DBconn2		
	TP (req/s)	CPU util (%)	L2 miss (×6000)	TP (req/s)	CPU util (%)	L2 miss (×6000)
1200	168	13.8	5036	169	13.6	4704
2400	340	34.9	8320	340	34.6	8153
3600	510	61.0	12304	510	60.2	11233
3800	538	**66.1**	**12963**	536	**64.5**	**11968**
4200	595	**76.6**	**14204**	595	**74.8**	**13053**
4600	642	**99.6**	**21868**	650	**86.2**	**14133**

Figure 7(b) shows the correlation between the number of L2 cache misses of MySQL and the corresponding CPU overhead from workload 1200 to 4600. The CPU overhead is calculated as shown in Figure 7(a) and the number of L2 cache misses in MySQL is recorded using the CPU performance counter [3] during the runtime experiments. This figure shows that the L2 cache misses and the corresponding CPU overhead are almost linearly correlated; thus higher L2 cache misses indicate higher CPU overhead.

One more interesting phenomenon we found is that the CPU overhead caused by L2 cache misses can be effectively reduced by limiting the concurrency level of request processing in the bottleneck server. Table 2 shows the comparison of CPU utilization and L2 cache

[3]The CPU performance counter increases by 1 for 6000 L2 cache misses in our environmental settings.

misses under two different DB connection pool sizes in Tomcat: DBconn12 and DBconn2. In the current RUBBoS implementation, each Servlet has its own local DB connection pool; DBconn12 means the DB connection pool size for each Servlet is 12 while DBconn2 means 2. This table shows that although the throughputs of these two cases are similar under different workloads, the DBconn2 case has less CPU utilization and less L2 cache misses in MySQL than the DBconn12 case, especially in the high workload range. We note that the DB connection pools in Tomcat controls the number of active threads in MySQL. In the DBconn12 case under high workload more concurrent requests are sent to the MySQL server, thus more concurrently active threads are created in MySQL and contend for the limited space of L2 cache causing more cache misses and CPU overhead than those in the DBconn2 case.

2.4.1.2 CPU overhead caused by Java GC

For Java-based servers like Tomcat and CJDBC, the JVM garbage collection process impacts the system response time fluctuations in two ways: first, the CPU time used by the garbage collector cannot be used for request processing; second, the JVM uses a synchronous garbage collector and it waits during the garbage collection period, only starting to process requests after the garbage collection is finished [5]. This delay significantly lengthens the pending requests and causes fluctuations in system response time.

Our measurements show that when a Java-based server is highly utilized, the JVM GCs of the server increase *nonlinearly* as workload increases. The hardware configuration of the experiments in this section is 1L/2L/1S/2L (see Figure 1(c)). Under this configuration, the CJDBC CPU is the bottleneck of the system. We note that the CJDBC server is a Java-based DB clustering middleware; each time a Tomcat server establishes a connection to the CJDBC server, which balances the load among the DB servers, a thread is created by CJDBC to route the SQL query to a DB server.

Table 3 compares the CPU utilization and the total GC time of the CJDBC server during the runtime experiments between the cases DBconn24 and DBconn2 from workload 3000 to 5600. This table shows that the total GC time for both the two cases increases nonlinearly

Table 3: Comparison of CJDBC CPU utilization and JVM GC time between DBconn24 and DBconn2 with 1L/2L/1S/2L configuration; higher concurrency leads to longer JVM GC time in the bottleneck tier (CJDBC).

WL	DBconn24			DBconn2		
	TP (req/s)	CPU util (%)	GC (s)	TP (req/s)	CPU util (%)	GC (s)
3000	428	49.6	0.05	428	49.2	0.05
4000	572	69.0	0.07	571	68.8	0.07
5000	721	**86.1**	**1.06**	719	**84.8**	**0.19**
5200	738	**91.2**	**1.51**	737	**87.4**	**0.37**
5400	759	**94.3**	**1.72**	767	**91.1**	**0.40**
5600	779	**98.8**	**2.15**	795	**96.6**	**0.45**

as workload increases, especially when the CJDBC CPU approaches saturation. One reason is that when the CJDBC CPU approaches saturation, the available CPU for GC shrinks; thus cleaning the same amount of garbage takes longer time than in the non-saturation situation. Accordingly, the impact of JVM GC on system response time fluctuations is more significant when CJDBC approaches saturation.

Table 3 also shows that the total GC time of the CJDBC server in the DBconn24 case is longer than that in the DBconn2 case from workload 5000 to 5600. The reason is similar to the L2 cache miss case as introduced in Section 2.4.1.1. Compared to the DBconn2 case, the Tomcat App tier in the DBconn24 case is able to send more concurrent requests to the CJDBC server under high workload, which in turn creates more concurrent threads for query routing and consumes more memory. Thus the CJDBC server performs more GCs for cleaning garbage in memory in the DBconn24 case than that in the DBconn2 case.

2.4.2 Fluctuation Amplification Effect in n-Tier Systems

Unlike some embarrassingly parallel "web indexing" applications using MapReduce and Hadoop, an n-tier application is unique in its amplification effect among different tiers due to the complex resource dependencies in the system. For instance, small request rate fluctuations from clients can be amplified to a bottom tier (e.g., DB tier), which causes significant response time fluctuation in the bottom tier; on the other hand, response time fluctuations in the bottom tier can be amplified to the front tiers.

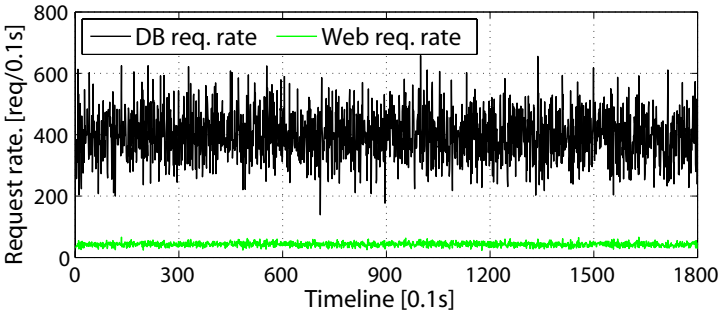


Figure 8: Amplified request rate flucutation from the web tier to the DB tier with 1L/2L/1L configuration in WL 3000.

Table 4: Statistic analysis of top-down request rate fluctuation amplification (corresponds to Figure 8).

Req. Rate (req/0.1s)	Web	App	DB
Mean	42.88	41.12	397.40
Std. Deviation	6.71	6.53	77.70
Coefficient of Variance.	**0.16**	0.16	**0.20**

2.4.2.1 Top-down request rate fluctuation amplification

The traffic for an n-tier system is, by nature, bursty [55]. One interesting phenomenon we found is that the bursty request rate from clients can be amplified to the bottom tier of the system. Except for the impact of transient events such as JVM GC, the complexity of inter-tier interactions of an n-tier system contributes most to the amplification effect. For example, a client's HTTP request may trigger multiple interactions between the application server tier and the DB tier to retrieve all the dynamic content to construct the web page requested by the client (We define the entire process as a client *transaction*).

Figure 8 shows the approximately instant request rate (aggregate at every 100ms) received by the Apache web tier and the MySQL DB tier of a three tier system (1L/2L/1L) in workload 3000. This figure shows that the request rate fluctuation in the MySQL tier is significantly larger than that in the Apache web tier. Table 4 shows the statistical analysis result of the amplification effect corresponding to Figure 8. This table shows three values

related to the request rate for each tier: mean, standard deviation, and coefficient of variation (CV) [4]. Comparing the mean request rate between the web tier and the DB tier, one HTTP request can trigger 9.3 database accesses on average, which explains why the instant DB request rate is much higher than the instant Web request rate; second, the CV of the request rate in the DB tier (0.20) is larger than that in the web tier (0.16), which shows the effect of request rate fluctuation amplification from the web tier to the DB tier.

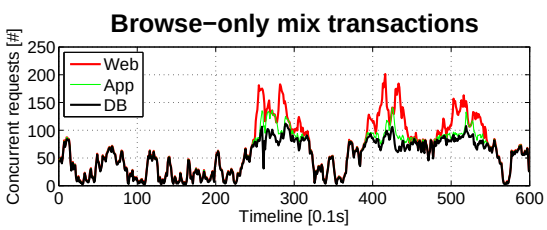
2.4.2.2 Bottom-up response time fluctuation amplification

Due to the top-down request rate fluctuation amplification and also the interference of transient events, the response time of the bottom tier in an n-tier system naturally fluctuates. We found that even small response time fluctuations in the bottom tier can be amplified to the front tiers due to the following two reasons.

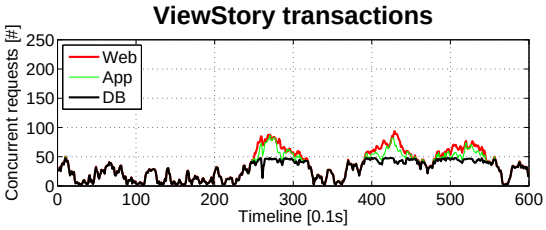
First, the complex soft resource dependencies among tiers may cause requests to queue in front tiers before they reach the bottom tier, which increases the waiting time of transaction execution. Soft resources refer to system software components such as threads, TCP connections, and DB connections [81]. In an n-tier system, every two consecutive tiers in an n-tier system are connected through soft resources during the long invocation chain of transaction execution in the system. For example, the Tomcat App tier connects to the MySQL tier through DB connections. Such connections are usually limited soft resources; once soft resources in a tier run out, the new requests coming to the tier have to queue in the tier until they get the released soft resources by other finished requests in the same tier. We note that for a RPC-style n-tier system, a request in a front tier releases soft resources (e.g., a processing thread) in the tier until the downstream tiers finish all the processing for the corresponding transaction. Accordingly, long response times in the bottom tier may lead to the saturation of soft resources (and thus a large number of queued requests) in front tiers.

Figure 9(a) shows the approximately instant number of concurrent requests (aggregated every 100ms) in each tier of a three-tier system (1L/2L/1L, MySQL is the bottleneck tier)

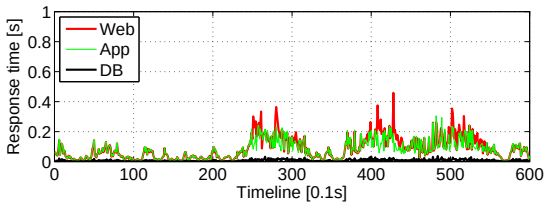
[4]Coefficient of variation means normalized standard deviation, which is standard deviation divided by mean.



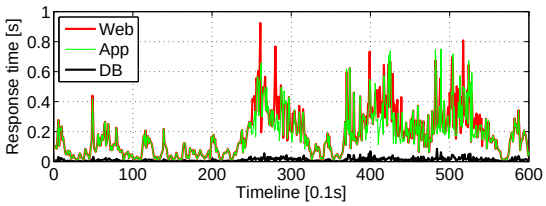
(a) Approximately instant # of concurrent requests in each tier



(b) Approximately instant # of concurrent ViewStory requests in each tier



(c) Approximately instant response time in each tier



(d) Approximately instant response time for ViewStory requests in each tier

Figure 9: Amplified response time fluctuations from the DB tier to the web tier with 1L/2L/1L (DBconn24) configuration in WL 5400.

under workload 5400. This figure shows that when the number of concurrent requests in MySQL reaches about 90, requests start to queue in the front tiers due to the scarcity of DB connections in Tomcat. Figure 9(c) shows the approximately instant response time in each tier. This figure shows that very small response time fluctuations (within 50ms) in MySQL lead to large response time fluctuations in Tomcat and Apache; the high peaks of response time in Figure 9(c) match well with the high peaks of queued requests in front tiers as shown in Figure 9(a). This indicates the waiting time of requests in front tiers largely contributes to the long response time of transaction execution.

Second, multi-interactions between tiers of an n-tier system amplify the bottom-up response time fluctuations. In an n-tier system it is natural that some transactions involve more interactions between different tiers than the other transactions. For example, in the RUBBoS benchmark, a ViewStory request triggers an average of twelve interactions between Tomcat and MySQL; a small response time increment in MySQL leads to a largely amplified response time in Tomcat and thus longer occupation time of soft resources in Tomcat. In such case, soft resources such as DB connections in Tomcat are more likely to run out, which leads to longer waiting time of the queued requests in Tomcat.

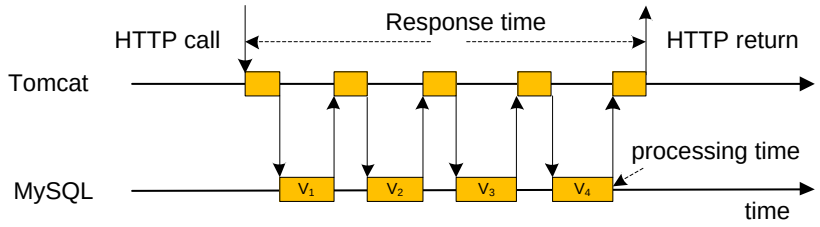
Figure 9(b) and 9(d) show the similar graphs as shown in Figure 9(a) and 9(c), but only for ViewStory transactions. Compared to Figure 9(c), Figure 9(d) shows that the response time of ViewStory requests in the Apache tier fluctuates more significantly. This is because ViewStory requests involve more interactions between Tomcat and MySQL than the average and run out their local DB connections earlier than the other types of requests; thus new incoming ViewStory requests have to wait longer in the Tomcat App tier (or in the Apache web tier if the connection resources between Apache and Tomcat also run out).

2.4.3 Mix-Transactions Scheduling in n-Tier Systems

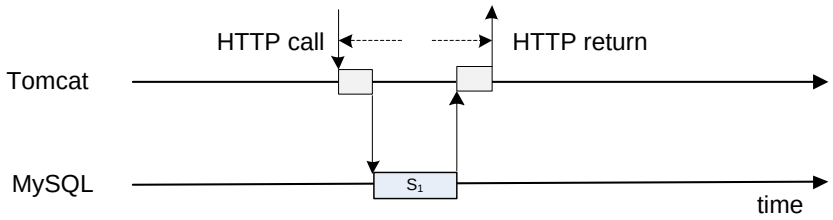
Scheduling polices impacting web server performance have been widely studied [44, 49, 71]. These previous works mainly focus on a single web server and show that the performance can be dramatically improved via a kernel-level modification by changing the scheduling policy from the standard FAIR (processor-sharing) scheduling to SJF (shortest-job-first) scheduling. However, for more complex n-tier systems where a completion of a client transaction involves complex interactions among tiers, the best OS level scheduling policy may increase the overall transaction response time.

The main reason for this is because the operating system of each individual server in an n-tier system cannot distinguish heavy transactions from light transactions without application level knowledge. A transaction being heavier than a light transaction can be caused by the heavy transaction having more interactions between different tiers than the light one. However, in each individual interaction the processing time of the involved tiers for a heavy transaction can be even smaller than that for a light transaction. Since the operating system of a tier can only schedule a job based on the processing time of the current interaction, applying SJF scheduling policy to the operating system of each tier may actually delay the application level light transactions.

Figure 10 shows sample interactions between a Tomcat App tier and a MySQL tier for a ViewStory transaction (heavy) and a StoryOfTheDay transaction (light) specified in the RUBBoS benchmark. A ViewStory transaction involves multiple interactions between Tomcat and MySQL (see Figure 10(a)) while a StoryOfTheDay transaction involves only one



(a) A sample ViewStory (heavy) transaction processing



(b) A sample StoryOfTheDay (light) transaction processing

Figure 10: ViewStory vs. StoryofTheDay, different interaction pattern between Tomcat and MySQL.

interaction (see Figure 10(b)). Suppose MySQL is the bottleneck tier. Our measurements show that a single query from a ViewStory transaction has similar execution time in MySQL as a query from a StoryOfTheDay transaction. During each interaction, a thread in the MySQL tier receives a query from Tomcat and returns a response after the query processing, regardless of which servlet sends the query. From MySQL's perspective, MySQL cannot distinguish which transaction is heavy and which transaction is light. Thus either FAIR or SJF scheduling in the MySQL tier can delay the processing of the light transactions.

We note that once the waiting time of queries from light transactions increases in MySQL, the total number of queued light requests in upper tiers also increases. Since each queued request (regardless if entailing heavy or light transactions) in upper tiers occupies soft resources such as threads and connections, soft resources in upper tiers are more likely to run out under high workload. In this case, the response time fluctuations in a bottom tier are more likely to amplified to upper tiers (see Section 2.4.2.2).

2.5 Related Work

Autonomic self-scaling n-tier systems based on elastic workload in cloud for both good performance and resource efficiency has been studied intensively before [50, 62, 83, 86]. The

main idea of these previous works is to propose adaptive control to manage application performance in cloud by combining service providers' SLA specifications (e.g., bounded response time) and virtual resource utilization thresholds. Based on the average of the monitored metrics (e.g., response time, CPU) over a period of time (a control interval), the controller of the system allocates necessary hardware resources to the bottleneck tier of the system once the target threshold is violated. However, how long a proper control interval should be is an open question and sometimes difficult to determine. As shown in this paper, the average of monitored metrics based on inappropriately long control intervals may blur the large performance fluctuations caused by factors such as bursty workload or JVM GC.

The performance impact of bursty workloads for the target n-tier system has been studied before. The authors in [24, 55] observed that it is important to consider different time scales when the system is under bursty workload; though the system CPU utilization may be low at a coarse time granularity, the CPU utilization fluctuates significantly if observed at a finer time granularity, and such large fluctuation significantly impacts the n-tier system response time. Different from the previous works which mainly focus on bursty workload, we focus more on system aspects such as JVM GC, scheduling policy, and fluctuation amplification effects in n-tier systems. As shown in this paper, the end-to-end response time presents large scale fluctuations because of system environmental conditions even under the moderately bursty workload.

Analytical models have been proposed for performance prediction and capacity planning of n-tier systems. Chen et al. [28] present a multi-station queuing network model with regression analysis to translate the service providers' SLA specifications to lower-level policies with the purpose of optimizing resource usage of an n-tier system. Thereska et al. [75] propose a queuing modeling architecture for clustered storage systems which constructs the model during the system design and continuously refines the model during operation for better accuracy due to the changes of system. Though these models have been shown to work well for particular domains, they are constrained by rigid assumptions such as normal/exponential distributed service times, disregard of some important factors inside the system which can cause significant fluctuations of both application and system level metrics.

2.6 Conclusions

We studied the large scale response time fluctuations of n-tier systems in high resource utilization using the n-tier benchmark RUBBoS. We found that the large scale response time fluctuations can be caused by some system environmental conditions such as L2 cache miss, JVM GC, and limitations of OS level scheduling policies in the system, in addition to the bursty workload from clients. We showed that because of the complex resource dependencies across tiers, a small response time fluctuation in a bottom tier can be amplified to front tiers and eventually to clients. To mitigate the large scale response time fluctuations, we evaluated three heuristics to reduce the latency long-tail problem while still achieving efficient resource utilization. Our work is an important contribution to design more effective autonomous self-scaling n-tier systems in cloud to achieve both good performance and resource efficiency under elastic workloads.

CHAPTER III

LINKING LATENCY LONG-TAIL TO TRANSIENT BOTTLENECKS

In the previous chapter we showed concrete experimental evidence of the presence of the latency long-tail problem and discussed several potential causes for the problem. In this chapter, we explicitly link the latency long-tail problem to the occurrence of transient bottlenecks. Applying micro-level event analysis on fine-grained measurement data from n-tier application benchmarks, we show that transient bottlenecks (from tens to hundreds of milliseconds) can cause queue overflows that propagate through an n-tier system, resulting in dropped messages and VLRT requests due to timeout and retransmissions. Our study shows that even at moderate CPU utilization levels, transient bottlenecks arise from several system layers, including Java garbage collection, anti-synchrony between workload bursts and DVFS clock rate adjustments, and statistical workload interferences among co-located VMs. We further build a simple model to quantify the negative impact of a transient bottleneck on system performance. Our simulation results show that we can avoid/mitigate the large response time variations caused transient bottlenecks by delimiting a safe utilization level of an n-tier web application.

3.1 Introduction

Wide response time fluctuations (latency long tail problem) of large scale distributed applications at moderate system utilization levels have been reported both in industry [32] and academia [47, 49, 78, 84]. Occasionally and without warning, some requests that usually return within a few milliseconds would take several seconds. These very long response time (VLRT) requests are difficult to study for two major reasons. First, the VLRT requests only take milliseconds when running by themselves, so the problem is not with the VLRT requests, but emerges from the interactions among system components. Second, the statistical average behavior of system components (e.g., average CPU utilization over typical measurement intervals such as minutes) shows all of them to be far from saturation.

Although our understanding of the VLRT requests has been limited, practical solutions to bypass the VLRT request problem have been described [32]. For example, applications with read-only semantics (e.g., web search) can use duplicate requests sent to independent servers and reduce perceived response time by choosing the earliest answer. These bypass techniques are effective in specific domains, contributing to an increasingly acute need to improve our understanding of the general *causes* for the VLRT requests. On the practical side, our lack of a detailed understanding of VLRT requests is consistent with the low average overall data center utilization [74] at around 18%, which is a more general way to avoid VLRT requests (see Section 3.4). The current situation shows that VLRT requests certainly merit further investigation and better understanding, both as an intellectual challenge and their potential practical impact (e.g., to increase the overall utilization and return on investment in data centers).

Using fine-grained monitoring tools (a combination of microsecond resolution message timestamping and millisecond system resource sampling), we have collected detailed measurement data on an n-tier benchmark (RUBBoS [8]) running in several environments. Micro-level event analyses show that VLRT requests can have very different causes, including CPU dynamic voltage and frequency scaling (DVFS) control at the architecture layer, Java garbage collection (GC) at the system software layer, and virtual machine (VM) consolidation at the VM layer. In addition to the variety of causes, the non-deterministic nature of VLRT requests makes the events dissimilar at the micro level.

Despite the wide variety of causes for VLRT requests, we show that they can be understood through the concept of *transient bottleneck*, defined as a very short period of time (on the order of tens of milliseconds), during which the CPU is saturated. When considered at the abstraction level of transient bottlenecks, the phenomenon of VLRT requests becomes reproducible: Even though the actual VLRT requests may not be literally the same ones, a similar number of VLRT requests arise reliably according to the timeline in experiments. Consequently, we are able to show concrete evidence that ties convincingly the various causes mentioned above to VLRT requests. We compare transient bottlenecks to lightning, since they are very short in duration (tens of milliseconds), but have a long impact on the

VLRT requests (several seconds).

The first contribution of the chapter is a set of micro-level event analyses of fine-grained experimental data of the RUBBoS n-tier benchmark in several environments. The initial steps of the micro-level event analyses are similar: (1) VLRT requests are detected of an n-tier system with moderate utilization; (2) at the same time, long request queues form in the Apache overflowing TCP buffer, that causing dropped packets and retransmission after 3 seconds; (3) the long queues in Apache formed because of downstream server (Tomcat) becoming saturated and incapable of servicing requests because Tomcat has completely full queues during that time; (4) long queues in Tomcat servers are created by transient bottlenecks, in which the server CPU becomes saturated for a very short period of time. We note that even though the transient bottlenecks are very short, the arrival rate of requests (thousands per second) quickly overwhelm the queues in the servers. The final step (5) of each micro-level event analysis identifies a specific cause associated with the transient bottlenecks: Java GC, DVFS, and VM consolidation.

The second contribution of the chapter is a simple generic model of the negative impact caused by transient bottlenecks in an n-tier web application. As we can see the essential problem of a transient bottleneck is the temporary resource shortage (e.g., temporary CPU saturation) during the bottleneck period, causing requests to queue in the bottlenecked server and potentially in the upper tiers due to inter-tier resource dependencies. Thus a transient bottleneck can be modeled as a temporary resource shortage causing requests to queue in the system, regardless of the root cause of the transient bottleneck. We studied how the duration of a temporary resource shortage impacts the end-to-end response time of an n-tier system at different utilization levels.

The rest of the paper is organized as follows. Section 3.2.2 shows the emergence of VLRT requests at increasing workload and utilization using the Java GC experiments. Section 3.3 describes the micro-level event analyses that link the varied causes to VLRT requests. Section 3.4 presents a simple generic model of the negative impact caused by transient bottlenecks. Section 3.5 summarizes the related work and Section 3.6 concludes the paper.

Software Stack

Web Server	Apache 2.0.54
Application Server	Apache Tomcat 5.5.17
Cluster middleware	C-JDBC 2.0.2
Database server	MySQL 5.0.51a
Sun JDK	jdk1.5.0_07, jdk1.6.0_14
Operating system	RHEL 6.2 (kernel 2.6.32)
Hypervisor	VMware ESXi v5.0
System monitor	esxtop 5.0, Sysstat 10.0.0

(a) Software setup

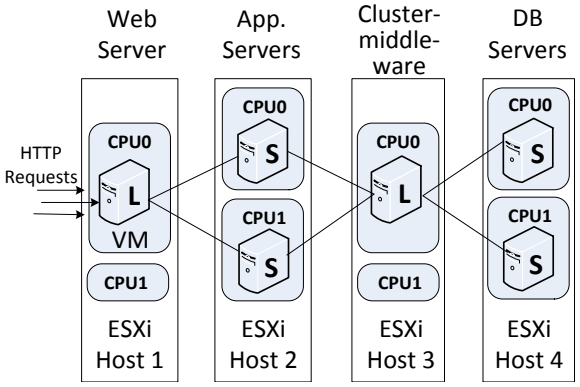
ESXi Host Configuration

Model	Dell Power Edge T410
CPU	2* Intel Xeon E5607, 2.26GHz Quad-Core
Memory	16GB
Storage	7200rpm SATA local disk

VM Configuration

Type	# vCPU	CPU limit	CPU shares	vRAM	vDisk
Large (L)	2	4.52GHz	Normal	2GB	20GB
Small (S)	1	2.26GHz	Normal	2GB	20GB

(b) ESXi host and VM setup



(c) 1L/2S/1L/2S sample topology

Figure 11: Details of the experimental setup.

3.2 Background and Motivation

3.2.1 Experimental Setup

We adopt the RUBBoS standard n-tier benchmark, based on bulletin board applications such as Slashdot [8]. RUBBoS can be configured as a three-tier (web server, application server, and database server) or four-tier (addition of clustering middleware such as C-JDBC [26]) system. The workload consists of 24 different web interactions, each of which is a combination of all processing activities that deliver an entire web page requested by a client, i.e., generate the main HTML file as well as retrieve embedded objects and perform related database queries. These interactions aggregate into two kinds of workload modes: browse-only and read/write mixes. We use browse-only workload in this paper. The closed-loop workload generator of this benchmark generates a request rate that follows a Poisson distribution parameterized by a number of emulated clients. Such workload generator has

a similar design as other standard n-tier benchmarks such as RUBiS, TPC-W, Cloudstone etc.

We run the RUBBoS benchmark on our virtualized testbed. Figure 11 outlines the software components, ESXi host and virtual machine (VM) configuration, and a sample topology used in the experiments. We use a four-digit notation $\#W/\#A/\#C/\#D$ to denote the number of web servers (Apache), application servers, clustering middleware servers (C-JDBC), and database servers. Figure 11(c) shows a sample 1/2/1/2 topology. Each server runs on top of one VM. Each ESXi host runs the VMs from the same tier of the application. The VMs from the same tier are pinned to separate CPU cores to reduce the interferences among VMs. We always deploy Apache and C-JDBC in type "L" VMs since we want to avoid bottlenecks in load-balance tiers. Hardware utilization measurements (e.g., CPU) are taken during the runtime period using Collectl [4] at 50ms granularity.

3.2.2 VLRT Requests at Moderate Utilization

Large response time fluctuations (also known as the latency long tail problem) of large scale distributed applications happen when very long response time (VLRT) requests arise. VLRT requests have been reported by industry practitioners [32] and academic researchers [47, 49, 78, 84]. These requests are difficult to study, since they happen occasionally and without warning, often at moderate CPU utilization levels. When running by themselves, the VLRT requests change back to normal and return within a few milliseconds. Consequently, the problem does not reside within the VLRT requests, but in the interactions among the system components.

Since VLRT requests arise from system interactions, usually they are not exactly reproducible at the request level. Instead, they appear when performance data are statistically aggregated, as their name "latency long tail" indicates. We start our study by showing one set of such aggregated graphs, using RUBBoS [8], a representative web-facing n-tier system benchmark modeled after Slashdot. Our experiments use a typical 4-tier configuration, with 1 Apache web server, 2 Tomcat Application Servers, 1 C-JDBC clustering middleware, and 2 MySQL database servers (details in Section 4.2.1).

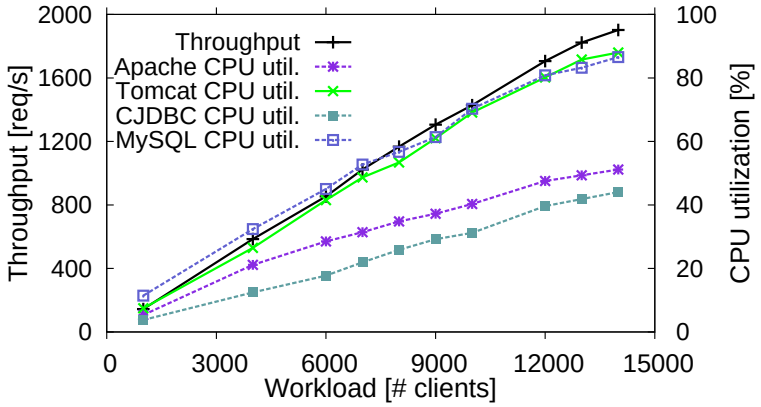


Figure 12: System throughput increases linearly with the CPU utilization of representative servers at increasing workload.

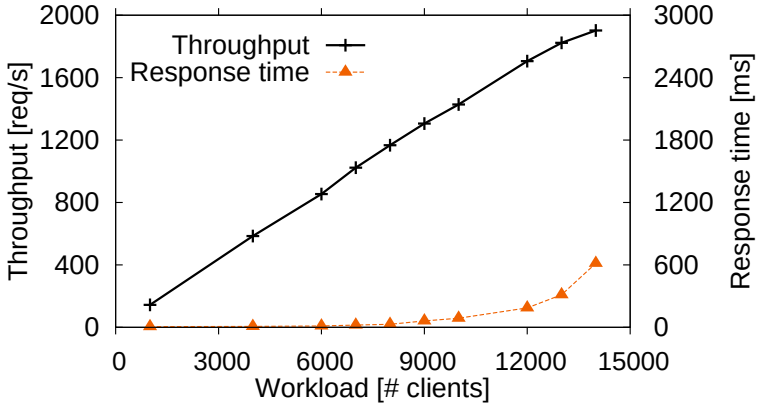


Figure 13: System throughput and average response time at increasing workload. The wide response time fluctuations are not apparent since the average response time is low (<200ms) before 12000 clients.

When looking at statistical average metrics such as throughput, VLRT requests may not become apparent immediately. As illustration, Figure 12 shows the throughput and CPU utilization of RUBBoS experiments for workloads from 1000 to 14000 concurrent clients. The average CPU utilization of Tomcat and MySQL rise gradually, as expected. The system throughput grows linearly, since all the system components have yet to reach saturation. Similarly, the aggregate response time graph (Figure 13) show little change up to 12000 clients. Without looking into the distribution of request response time, one might overlook the VLRT problems that start at moderate CPU utilization levels.

Although not apparent from Figure 12, the percentage of VLRT requests (defined as requests that take more than 3 seconds to return in this paper) increases significantly

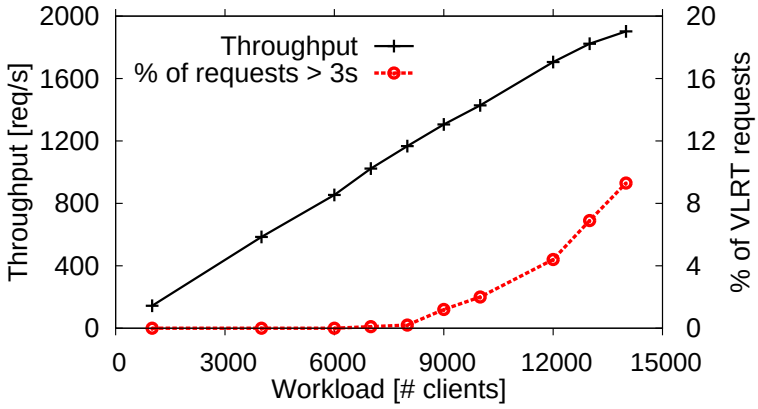
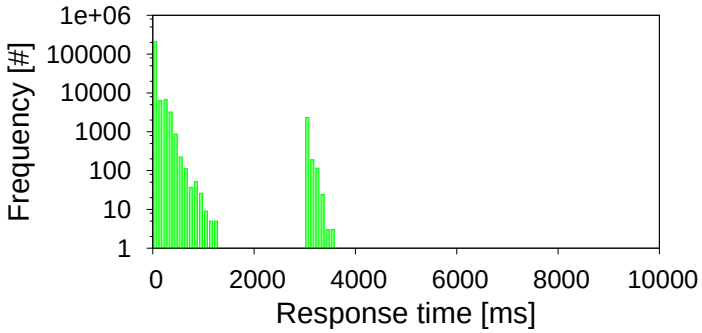


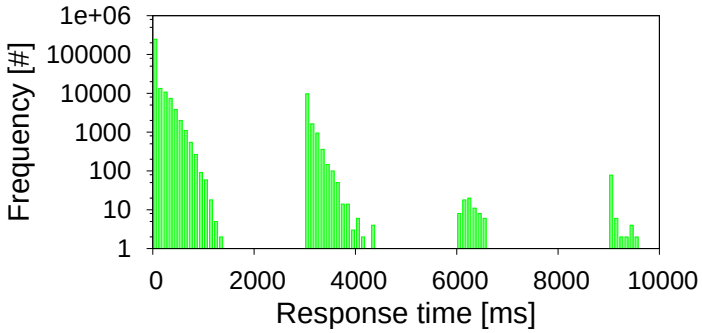
Figure 14: The percentage of VLRT requests starts to grow rapidly starting from 9000 clients.

starting from 9000 clients as shown in Figure 14. At the workload of 12000 clients, more than 4% of all requests become VLRT requests, even though the CPU utilization of all servers is only 80% (Tomcat and MySQL) or much lower (Apache and C-JDBC). The latency long tail problem can be seen more clearly when we plot the frequency of requests by their response times in Figure 15 for two representative workloads: 9000 and 12000. At moderate CPU utilization (about 61% at 9000 clients, Figure 15(a)), VLRT requests appear as a second cluster after 3 seconds. At moderately high CPU utilization (about 81% at 12000 clients, Figure 15(b)), we see 3 clusters of VLRT requests after 3, 6, and 9 seconds, respectively. These VLRT requests add up to 4% as shown in Figure 14.

One of the intriguing (and troublesome) aspects of wide response time fluctuations is that they start to happen at moderate CPU utilization level (e.g., 61% at 9000 clients). This observation suggests that the CPU (the critical resource) may be saturated only part of the time, which is consistent with previous work [78, 80] on transient bottlenecks as potential causes for the VLRT requests. Complementing a technical problem-oriented description of transient bottlenecks (Java garbage collection [78] and anti-synchrony from DVFS [80]), we also show that VLRT requests are associated with a more fundamental phenomenon (namely, *transient bottleneck*) that can be described, understood, and remedied in a more general way than each technical problem.



(a) 9000 clients; the system throughput is 1306 req/s and the highest average CPU usage among component servers is 61%.



(b) 12000 clients; the system throughput is 1706 req/s and the highest average CPU usage among component servers is 81%.

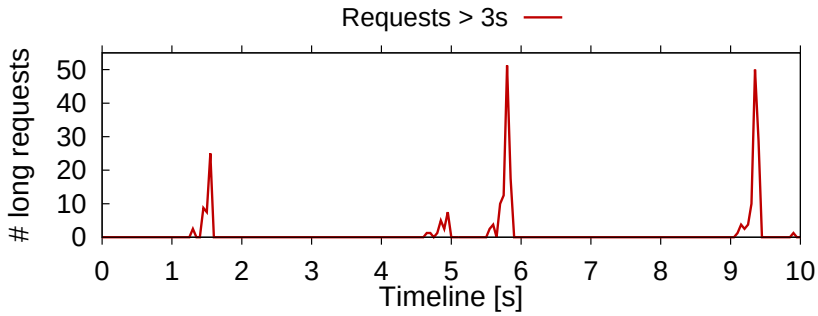
Figure 15: Frequency of requests by their response times at two representative workloads. The system is at moderate utilization, but the latency long tail problem can be clearly seen.

3.3 VLRT Requests Caused by Transient Bottlenecks

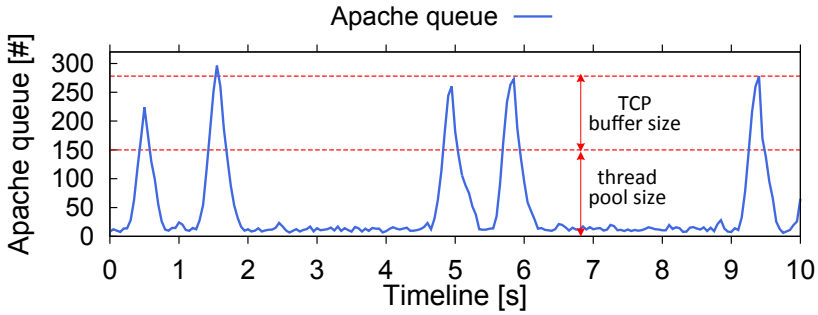
We use a micro-level event analysis to link the causes of transient bottlenecks to VLRT requests. The micro-level event analysis exploits the fine-grained measurement data collected in RUBBoS experiments. Specifically, all messages exchanged between servers are timestamped at microsecond resolution. In addition, system resource utilization (e.g., CPU) is monitored at short time intervals (e.g., 50ms). The events are shown in a timeline graph, where the X-axis represents the time elapsed during the experiment at fine-granularity (50ms units in this section).

3.3.1 VLRT Requests Caused by Java GC

In our first illustrative case study of transient bottlenecks, we will establish the link between VLRT requests shown in Figure 15 to the Java garbage collector (GC) in the Tomcat



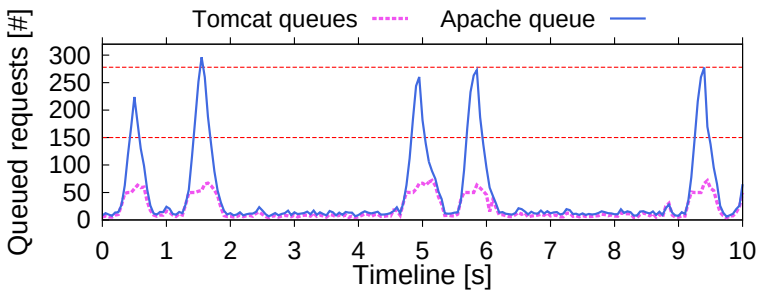
(a) Number of VLRT requests counted at every 50ms time window. Such VLRT requests contribute to bi-modal response time distribution as shown in Figure 15(a).



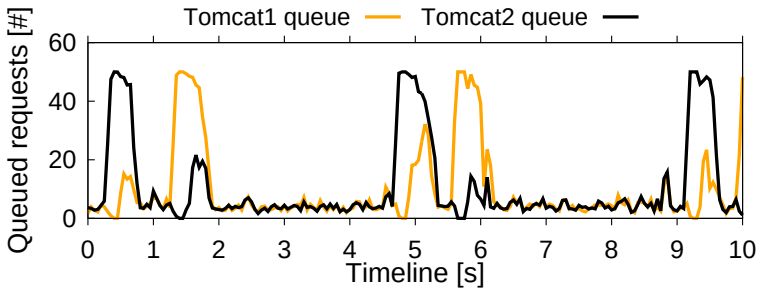
(b) Frequent queue peaks in Apache during the same 10-second time-frame as in (a). The queue peaks match well with the occurrence of the VLRT requests in (a). This arises because Apache drops new incoming packets when the queued requests exceed the upper limit of the queue, which is imposed by the server thread pool size (150) and the operating system TCP stack buffer size (128 by default). Dropped packets lead to TCP retransmissions (>3s).

Figure 16: VLRT requests (see (a)) caused by queue peaks in Apache (see (b)) when the system is at workload 9000 clients.

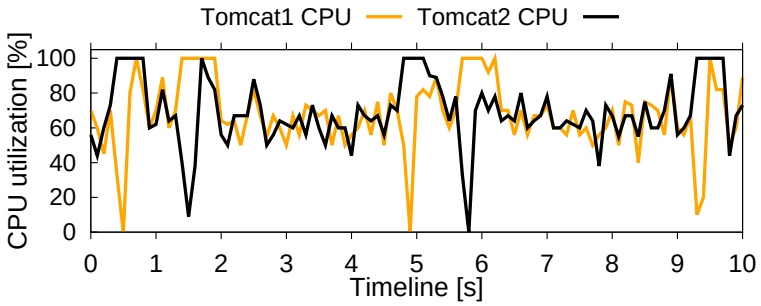
application server tier of the n-tier system. We have chosen Java GC as the first case because it is deterministic and easier to explain. Although Java GC has been suggested as a cause of transient events [78], the following explanation is the first detailed description of data flow and control flow that combine into queue amplification in an n-tier system. This description is a five-step micro-event timeline analysis of fine-grained monitoring based on a system tracing facility that timestamps all network packets at microsecond granularity [80]. By recording the precise arrival and departure timestamps of each client request for each server, we are able to determine precisely how much time each request spends in each tier of the system.



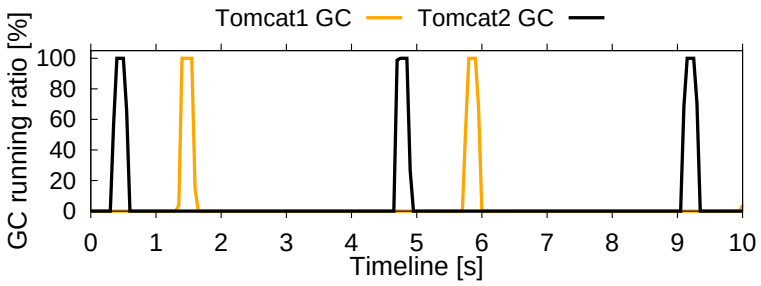
(a) Queue peaks in Apache coincide with the queue peaks in Tomcat, suggesting push-back wave from Tomcat to Apache.



(b) Request queue for each Tomcat server (1 and 2). The sum of the two is the queued requests in the Tomcat tier (see (a)).



(c) Transient CPU saturations of a Tomcat server coincide with the queue peaks in the corresponding Tomcat server (see (b)).



(d) Episodes of Java GC in a Tomcat server coincide with the transient CPU saturation of the Tomcat server (see (c)).

Figure 17: Queue peaks in Apache (a) due to transient bottlenecks caused by Java GC in Tomcat (d).

In the first step of micro-event analysis (transient events), we use fine-grained monitoring data to determine which client requests are taking seconds to finish instead of the normally expected milliseconds response time. Specifically, we know exactly at what time these VLRT requests occur. A non-negligible number (up to 50) of such VLRT requests appear reliably (even though they may not be exactly the same set for different experiments) at approximately every four seconds as measured from the beginning of each experiment (Figure 16(a)). The X-axis of Figure 16(a) is a timeline at 50ms intervals, showing the clusters of VLRT requests are tightly grouped within a very short period of time. Figure 16(a) shows four peak/clusters of VLRT requests during a 10-second period of a RUBBoS experiment (workload 9000 clients). Outside of these peaks, all requests return within milliseconds, consistent with the average CPU utilization among component servers being equal to or lower than 61%.

In the second step of micro-event analysis (retransmitted requests), we show that dropped message packets are likely the cause of VLRT requests. To make this connection, we first determine which events are being queued in each server. In an n-tier system, we say that a request is waiting in a queue at a given tier when its request packet has arrived and a response has not been returned to an upstream server or client. This situation is the n-tier system equivalent of having a program counter entering that server but not yet exited. Using the same timeframe of Figure 16(a), we plot the request queue length in the Apache server in Figure 16(b). Figure 16(b) shows five peak/clusters, in which the number of queued requests in Apache is higher than 150 for that time interval. The upper limit of the queued requests is slightly less than 300, which is comparable to the sum of thread pool size (150 threads) plus TCP buffer size of 128. Although there is some data analysis noise due to the 50ms window size, the number of queued requests in Apache suggests strongly that some requests may have been dropped, when the thread pool is entirely consumed (using one thread per incoming request) and then the TCP buffer becomes full. Given the 3-second retransmission timeout for TCP (kernel 2.6.32), we believe the overlapping peaks of Figure 16(a) (VLRT requests) and Figure 16(b) (queued requests in Apache) make a convincing case for dropped TCP packets causing the VLRT requests. However, we still

need to find the source that caused the requests to queue in the Apache server, since Apache itself is not a bottleneck (none of the Apache resources is a bottleneck).

In the third step of micro-event analysis (queue amplification), we continue the per-server queue analysis by integrating and comparing the requests queued in Apache Figure 16(b) with the requests queued in Tomcat. The five major peak/clusters in Figure 17(a) show the queued requests in both Apache (sharp/tall peaks near the 278 limit) and Tomcat (lower peaks within the sharp/tall peaks). This near-perfect coincidence of (very regular and very short) queuing episodes suggests that it is not by chance, but somehow Tomcat may have contributed to the queued requests in Apache.

Let us consider more generally the situation in n-tier systems where queuing in a downstream server (e.g., Tomcat) is associated with queuing in the upstream server (e.g., Apache). In client/server n-tier systems, a client request is sent downstream for processing, with a pending thread in the upstream server waiting for the response. If the downstream server encounters internal processing delays, two things happen. First, the downstream server's queue grows. Second, the number of matching and waiting threads in the upstream server also grows due to the lack of responses from downstream. This phenomenon, which we call *push-back wave*, appears in Figure 17(a). The result of the third step in micro-event analysis is the connection between long queue in Apache to queuing in Tomcat due to Tomcat saturation.

In the fourth step of micro-event analysis (transient bottlenecks), we will link the Tomcat queuing with transient bottlenecks in which CPU becomes saturated for a very short time (tens of milliseconds). The first part of this step is a more detailed analysis of Tomcat queuing. Specifically, the queued requests in the Tomcat tier (a little higher than 60 in Figure 17(a)), are the sum of two Tomcat servers. The sum is meaningful since a single Apache server uses the two Tomcat servers to process the client requests. To study the transient bottlenecks of CPU, we will consider the request queue for each Tomcat server (called 1 and 2) separately in Figure 17(b). At about 0.5 seconds in Figure 17(b), we can see Tomcat2 suddenly growing a queue that contains 50 requests, due to the concurrency limit of the communication channel between each Apache process and a Tomcat instance

(AJP [3] connection pool size set to 50).

The second part of the fourth step is a fine-grained sampling (at 50ms intervals) of CPU utilization of Tomcat, shown in Figure 17(c). We can see that Tomcat2 enters a full (100%) CPU utilization state, even though it is for a very short period of about 300 milliseconds. This short period of CPU saturation is the transient bottleneck that caused the Tomcat2 queue in Figure 17(b) and through push-back wave, the Apache queue in Figure 17(a)). Similar to the Tomcat2 transient bottleneck at 0.5 seconds in Figure 17(b), we can see a similar Tomcat1 transient bottleneck at 1.5 seconds. Each of these transient bottlenecks is followed by similar bottlenecks every four seconds during the entire experiment.

The fifth step of the micro-event analysis (root cause) is the linking of transient CPU bottlenecks to Java GC episodes. Figure 17(d) shows the timeline of Java GC, provided by the JVM GC logging. We can see that both Tomcat1 and Tomcat2 run Java GC at a regular time interval of about four seconds. The timeline of both figures shows that the transient bottlenecks in Figure 17(c) and Java GC episodes happen at the same time throughout the entire experiment. The experiments were run with JVM 1.5, which is known to consume significant CPU resources at high priority during GC. This step shows that the Java GC caused the transient CPU bottlenecks.

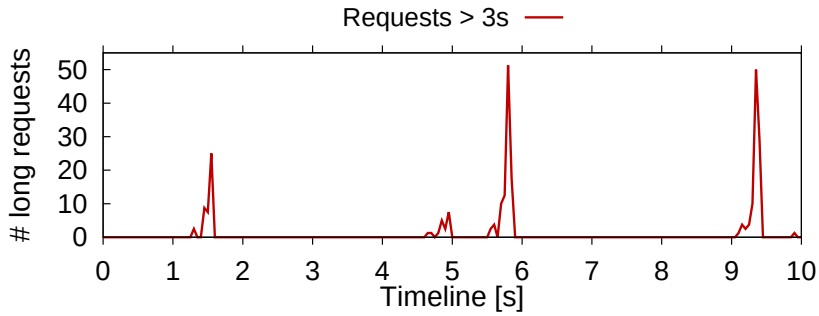
In summary, the 5 steps of micro-event analysis show the VLRT requests in Figure 15 are due to transient bottlenecks caused by Java GC:

1. Transient events: VLRT requests are clustered within a very short period of time at about 4-second intervals throughout the experiment (Figure 16(a)).

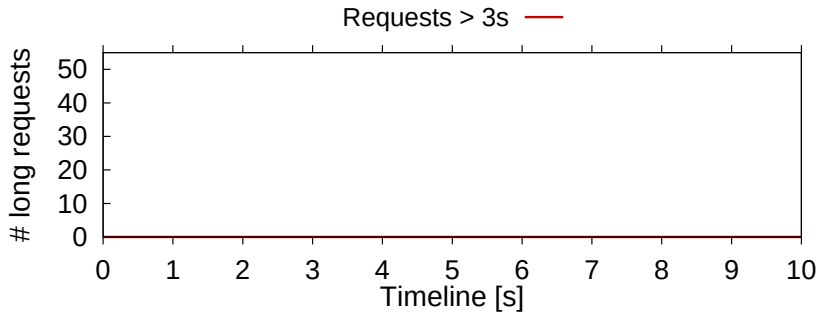
2. Retransmitted requests: VLRT requests coincide with long request queues in the Apache server (Figure 16(b)) that causes dropped packets and TCP retransmission after 3 seconds.

3. Queue amplification: long queues in Apache are caused by push-back waves from Tomcat servers, where similar long queues form at the same time (Figure 17(a)).

4. Transient bottlenecks: long queues in Tomcat (Figure 17(b)) are created by transient bottlenecks (Figure 17(c)), in which the Tomcat CPU becomes saturated for a very



(a) JDK 1.5 case; the number of VLRT requests measured during a 10-second time period.



(b) JDK 1.6 case; the number of VLRT requests measured during a 10-second time period.

Figure 18: VLRT requests caused by Java GC can be solved by upgrading JDK from 1.5 to 1.6.

short period of time (about 300 milliseconds).

5. Root cause: The transient bottlenecks coincide exactly with Java GC episodes (Figure 17(d)).

The transient bottlenecks caused by Java GC can be solved by upgrading the JDK version in Tomcat from 1.5 to 1.6 (see Figure 18). This is because JDK 1.6 uses a much more efficient garbage collector and reduces the demands on CPU [5], thus avoiding the transient bottlenecks due to Java GC. In the following subsections we always use JDK 1.6 in Tomcat and show the transient bottlenecks caused by other factors.

3.3.2 VLRT Requests Caused by Anti-Synchrony from DVFS

The second case of transient bottlenecks was found to be associated with anti-synchrony between workloads bursts and CPU clock rate adjustments made by dynamic voltage and frequency scaling (DVFS). By anti-synchrony we mean opposing cycles, e.g., CPU clock

Table 5: Percentage of VLRT requests and the resource utilization of representative servers as workload increases in the SpeedStep case.

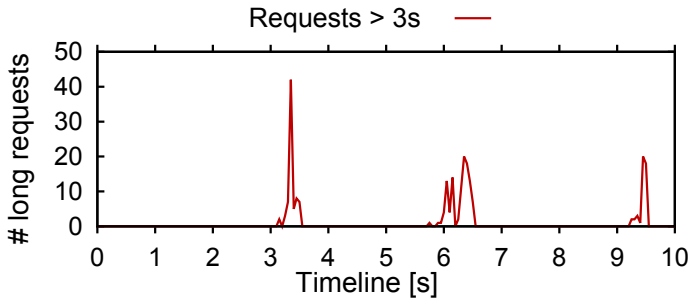
Workload	6000	8000	10000	12000
requests > 3s	0	0.3%	0.2%	0.7%
Tomcat CPU util.	31%	43%	50%	61%
MySQL CPU util.	44%	56%	65%	78%

rate changed from high to low after idling, but the slow CPU immediately meets a burst of new requests. Previous work [32, 79] have suggested power saving techniques such as DVFS being a potential source for VLRT requests. The following micro-event analysis will explain in detail the queue amplification process that links anti-synchrony to VLRT requests through transient bottlenecks.

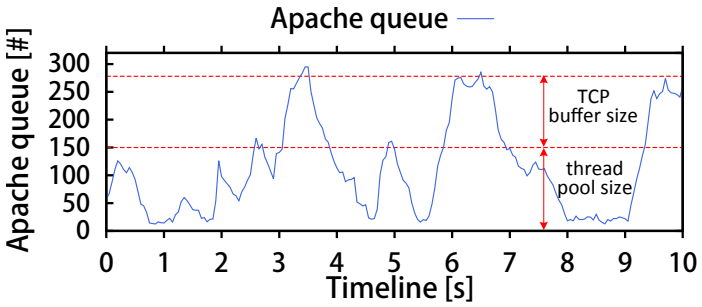
In DVFS experiments, VLRT requests start to appear at 8000 clients (Table 5) and grow steadily with increasing workload and CPU utilization, up to 0.7% of all requests at 12000 clients with 78% CPU in MySQL. These experiments (similar to [79]) had the same setup as Java GC experiments in Section 3.3.1, with two modifications. First, the JDK in Tomcat was upgraded from 1.5 to 1.6 to avoid the transient bottlenecks described in Section 3.3.1 due to Java GC. Second, the DVFS control (default Dell BIOS level) in MySQL is turned on: Intel Xeon CPU (E5607) supporting nine CPU clock rates, with the slowest (P8, 1.12 GHz) nearly half the speed of the highest (P0, 2.26GHz).

In the first step of micro-event analysis (transient events) for DVFS experiments, we plot the occurrence of VLRT requests (Figure 19(a)) through the first 10-second of experiment with workload of 12000 clients. Three tight clusters of VLRT requests appear, showing the problem happened during a very short period of time. Outside of these tight clusters, all requests return within a few milliseconds.

In the second step of micro-event analysis (dropped requests), the request queue length in Apache over the same period of time shows a strong correlation between peaks of Apache queue (Figure 19(b)) and peaks in VLRT requests (Figure 19(a)). Furthermore, the three high Apache queue peaks rise to the sum of Apache thread pool size (150) and its TCP buffer size (128). This observation is consistent with the first illustrative case, suggesting



(a) Number of VLRT requests counted at every 50ms time window.

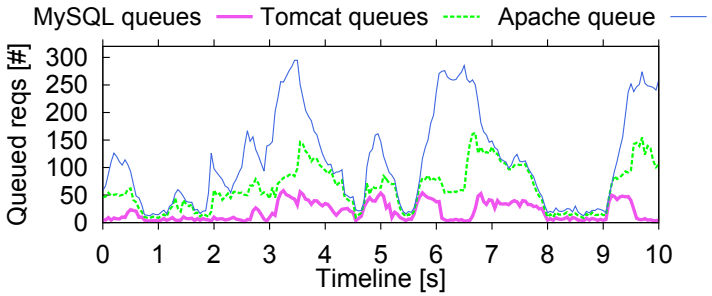


(b) Frequent queue peaks in Apache during the same 10-second time period as in (a). Once a queue spike exceeds the concurrency limit, new incoming packets are dropped and TCP retransmission occurs, causing the VLRT requests as shown in (a).

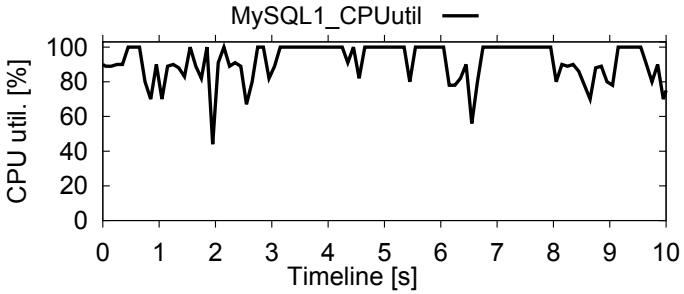
Figure 19: VLRT requests (see (a)) caused by queue peaks in Apache (see (b)) when the system is at workload 12000.

dropped request packets during those peak periods, even though Apache is very far from saturation (46% utilization).

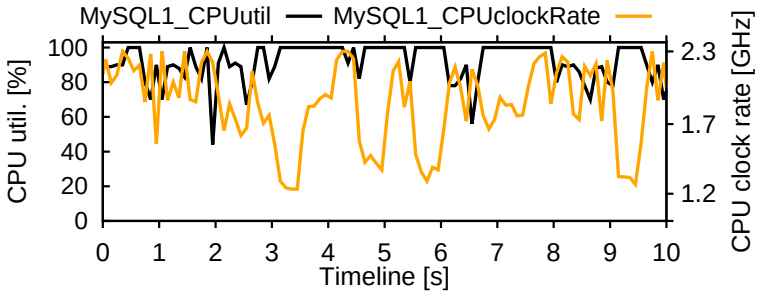
In the third step of micro-event analysis (queue amplification), we establish the link between the queuing in Apache with the queuing in downstream servers by comparing the queue lengths of Apache, Tomcat, and MySQL in Figure 20(a). We can see that peaks of Apache queue coincide with peaks of queue lengths in Tomcat and MySQL. A plausible hypothesis is queue amplification that starts in MySQL, propagating to Tomcat, and ending in Apache. Supporting this hypothesis is the height of queue peaks for each server. MySQL has 50-request peaks, which is the maximum number of requests sent by Tomcat, with database connection pool size of 50. Similarly, a Tomcat queue is limited by the AJP connection pool size in Apache. As MySQL reaches full queue, a push-back wave starts to fill Tomcat's queues, which propagates to fill Apache's queue. When Apache's queue becomes full, dropped request messages create VLRT requests.



(a) Queue peaks in Apache coincide with the queue peaks in MySQL, suggesting the push-back waves from MySQL to Apache.



(b) Transient CPU saturation periods of MySQL1 coincide with the queue peaks in MySQL (see (a)).



(c) The low CPU clock rate of MySQL1 coincides with the transient CPU saturation periods, suggesting that the transient CPU saturation is caused by the delay of CPU adapting from a slow mode to a faster mode to handle a workload burst.

Figure 20: Queue peaks in Apache (see (a)) due to transient bottlenecks in MySQL caused by the anti-synchrony between workload bursts and DVFS CPU clock rate adjustments (see (c)).

In the fourth step of micro-event analysis (transient bottlenecks), we will link the MySQL queue to transient bottlenecks with a fine-grained CPU utilization plot of MySQL server (Figure 20(b)). A careful comparative examination of Figure 20(b) and Figure 20(a) shows that short periods of full (100%) utilization of MySQL coincide with the same periods where MySQL reaches peak queue length (the MySQL curve in Figure 20(a)). For simplicity, Figure 20(b) shows the utilization of one MySQL server, since the other MySQL shows the same correlation.

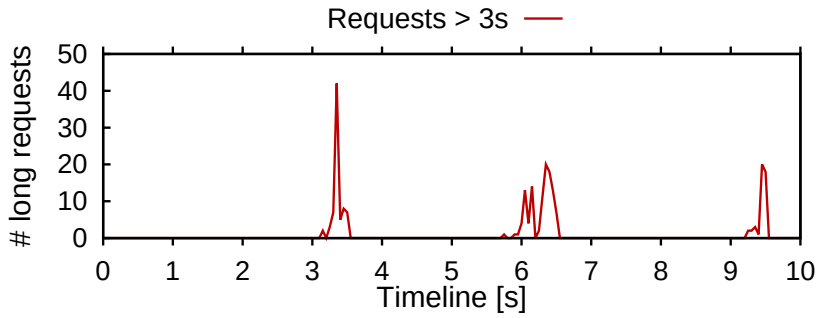
The fifth step of the micro-event analysis (root cause) is the linking of transient CPU bottlenecks to the anti-synchrony between workload bursts and CPU clock rate adjustments. The plot of CPU utilization and clock rate of MySQL server shows that CPU saturation leads to a rise of clock rate and non-saturation makes the clock rate slow down (Figure 20(c)). While this is the expected and appropriate behavior of DVFS, a comparison of Figure 20(a), Figure 20(b), and Figure 20(c) shows that the MySQL queue tends to grow while clock rate is slow (full utilization), and fast clock rates tend to empty the queue and lower utilization. Anti-synchrony becomes a measurable issue when the DVFS adjustment periods (500ms in Dell BIOS) and workload bursts (default setting of RUBBoS) have similar cycles, causing the CPU to be in the mismatched state (e.g., low CPU clock rate with high request rate) for a significant fraction of time.

In summary, the 5 steps of micro-event analysis show the VLRT requests in Figure 19(a) are due to transient bottlenecks caused by the anti-synchrony between workload bursts and DVFS CPU clock rate adjustments:

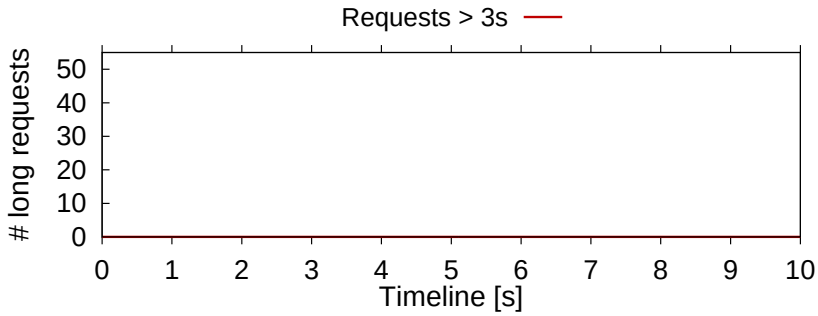
1. Transient events: VLRT requests are clustered within a very short period of time (three times in Figure 19(a)).

2. Retransmitted requests: VLRT requests coincide with periods of long request queues that form in the Apache server (Figure 19(b)) causing dropped packets and TCP retransmission.

3. Queue amplification: The long queues in Apache are caused by push-back waves from MySQL and Tomcat, where similar long queues form at the same time (Figure 20(a)).



(a) The DVFS ON case; the number of VLRT requests measured during a 10-second time period.



(b) The DVFS OFF case; the number of VLRT requests measured during a 10-second time period.

Figure 21: VLRT requests caused by anti-synchrony between workload bursts and DVFS CPU clock rate adjustments can be avoid by turning off DVFS.

4. Transient bottlenecks: The long queue in MySQL (Figure 20(a)) is created by transient bottlenecks (Figure 20(b)), in which the MySQL CPU becomes saturated for a short period of time (ranging from 300 milliseconds to slightly over 1 second).

5. Root cause: The transient bottlenecks are caused by the anti-synchrony between workload bursts and DVFS CPU clock rate adjustments (Figure 20(c)).

The transient bottlenecks caused by anti-synchrony between workload bursts and DVFS CPU clock rate adjustments can be avoid by turning off DVFS (see Figure 21). This is because in the DVFS off case MySQL CPU always operates at the maximum CPU clock rate, thus avoiding the transient bottlenecks due to the anti-synchrony. In the following subsection we turn off DVFS in all the servers and continue to show transient bottlenecks caused by other factors.

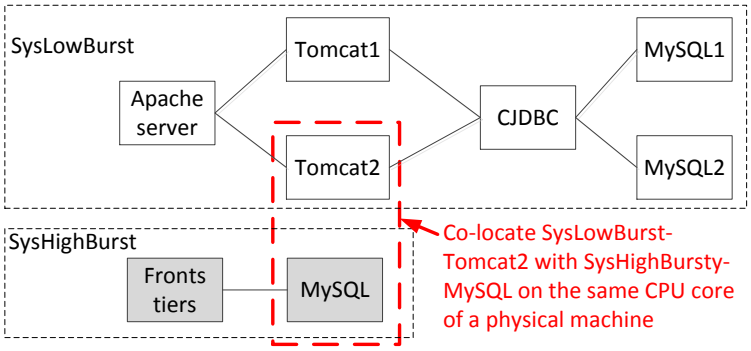


Figure 22: Consolidation strategy between SysLowBurst and SysHighBurst; the Tomcat2 in SysLowBurst is co-located with MySQL in SysHighBurst.

3.3.3 VLRT Requests Caused by Interferences among Consolidated VMs

The third case of transient bottlenecks was found to be associated with the interferences among consolidated VMs. VM consolidation is an important strategy for cloud service providers to share infrastructure costs and increase profit [20, 42]. An illustrative win-win scenario of consolidation is to co-locate two independent VMs with bursty workloads [54] that do not overlap, so the shared physical node can serve each one well and increase overall infrastructure utilization. However, statistically independent workloads tend to have somewhat random bursts, so the bursts from the two VMs sometimes alternate, and sometimes overlap. The interferences among co-located VMs is also known as the "noisy neighbors" problem. The following micro-event analysis will explain in detail the queue amplification process that links the interferences among consolidated VMs to VLRT requests through transient bottlenecks.

The experiments that study the interferences between two consolidated VMs consist of two RUBBoS n-tier applications, called SysLowBurst and SysHighBurst (Figure 22). SysLowBurst is very similar to the 1/2/1/2 configuration of previous experiments on Java VM and DVFS (Sections 3.3.1 and 3.3.2), while SysHighBurst is a simplified 1/1/1 configuration (one Apache, one Tomcat, and one MySQL). The only shared node runs VMware ESXi, with the Tomcat in SysLowBurst co-located with MySQL in SysHighBurst on the same CPU core. All other servers run on dedicated nodes. The experiments use JVM 1.6 and CPUs with disabled DVFS, to eliminate those two known causes of transient bottlenecks.

Table 6: Workload of SysLowBurst and SysHighBurst during consolidation. SysLowBurst is serving 14000 clients with burstiness $I = 1$ and SysHighBurst is serving 400 clients but with increasing burstiness levels. As the burstiness of SysHighBurst's workload increases, the percentage of VLRT requests in SysLowBurst increases.

#	SysLowBurst			SysHighBurst		
	WL	requests > 3s	Tomcat2-CPU (%)	WL	Burstiness level	MySQL-CPU (%)
1	14000	**0**	74.1	0	Null	0
2	14000	**0.1%**	74.9	400	I=1	10.2
3	14000	**2.7%**	74.7	400	I=100	10.6
4	14000	**5.0%**	75.5	400	I=200	10.5
5	14000	**7.5%**	75.2	400	I=400	10.8

The experiments evaluate the influence of bursty workloads by using the default RUBBoS workload generator (requests generated following a Poisson distribution parameterized by the number of clients) in SysLowBurst, and observing the influence of increasingly bursty workload injected by SysHighBurst. The workload generator of SysHighBurst is enhanced with an additional burstiness control [55], called *index of dispersion* (abbreviated as I). The workload burstiness $I = 1$ is calibrated to be the same as the default RUBBoS setting, and a larger I generates a burstier workload (for each time window, wider variations of requests created).

The baseline experiment runs SysLowBurst by itself at a workload of 14000 clients (no consolidation), with the result of zero VLRT requests (Table 6, line #1). The consolidation is introduced by SysHighBurst, which has a very modest workload of 400 clients, which is about 3% of SysLowBurst. However, the modest workload of SysHighBurst has an increasing burstiness from $I = 1$ to $I = 400$, when most of SysHighBurst workload become batched into short bursts. The lines #2 through #5 of Table 6 shows the increasing number of VLRT requests as I increases. We now apply the micro-event timeline analysis to confirm our hypothesis that the VLRT requests are caused by the interferences between the Tomcat2 in SysLowBurst and MySQL in SysHighBurst.

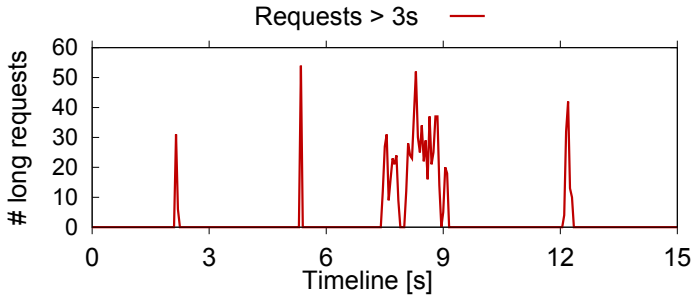
In the first step of micro-event analysis (transient events), we plot the occurrence of the VLRT requests of SysLowBurst (Figure 23(a)) during a 15-second of experiment when the consolidated SysHighBurst has $I = 100$ bursty workload. We can see three tight clusters (at 2, 5, and 12 seconds) and 1 broader cluster (around 9 seconds) of VLRT requests appear,

showing the problem happened during a relatively short period of time. Outside of these tight clusters, all requests return within a few milliseconds.

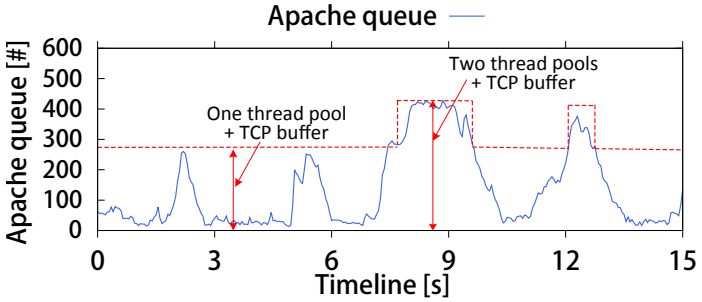
In the second step of micro-event analysis (dropped requests), we found the request queue in the Apache server of SysLowBurst has grown (Figure 23(b)) at the same time as the VLRT requests' peak times (Figure 23(a)). We will consider the two earlier peaks (at 2 and 5 seconds) first. These peaks (about 278, sum of thread pool size and TCP buffer size) are similar to the corresponding previous figures (Figure 16(b) and 19(b)), where requests are dropped due to Apache thread pool being consumed, followed by TCP buffer overflow. The two later peaks (centered around 9 and 12 seconds) are higher (more than 400), reflecting the creation of a second Apache process with another set of thread pools (150). The second process is spawned only when the first thread pool is fully used for some time. We found that packets get dropped during the higher peak periods for two reasons: during the initiation period of the second process (using non-trivial CPU resources, although for a very short time) and after the entire second thread pool has been consumed in a situation similar to earlier peaks.

In the third step of micro-event analysis (queue amplification), we establish the link between queues in Apache with queues in downstream servers by comparing the queue lengths of Apache and Tomcat in Figure 24(a). We can see that the four peaks in Tomcat coincide with the queue peaks in Apache (reproduced from the previous figure), suggesting that queues in Tomcat servers have contributed to the growth of queued requests in Apache, since the response delays would prevent Apache to continue. Specifically, the maximum number of requests between each Apache process and each Tomcat is the AJP connection pool size (50 in our experiments). As each Apache process reaches its AJP connection pool size and TCP buffer filled, newly arrived packets are dropped and retransmitted, creating VLRT requests.

In the fourth step of micro-event analysis (transient bottlenecks), we will link the Tomcat queues with the transient bottlenecks in which CPU becomes saturated for a very short period (Figure 24(b)). We can see that the periods of CPU saturation in Tomcat of SysLowBurst coincide with the Tomcat queue peaks (the Tomcat curve in Figure 24(a)), suggesting



(a) Number of VLRT requests counted at every 50ms time window.



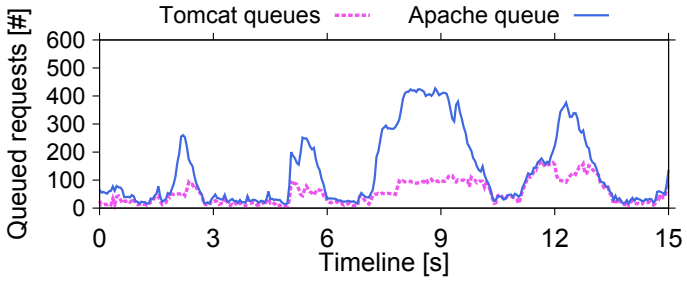
(b) Queue peaks in Apache coincide with the occurrence of the clustered VLRT requests (see (a)), suggesting those VLRT requests are caused by the queue peaks in Apache. Different from Figure 16(b) and 19(b), the Apache server here is configured to have two processes, each of which has its own thread pool. The second process is spawned only when the first thread pool is fully used. However, requests still get dropped when the first thread pool and the TCP buffer are full (at time marker 2 and 5).

Figure 23: VLRT requests (see (a)) caused by queue peaks in Apache (see (b)) in SysLowBurst when the consolidated SysHighBurst is at $I = 100$ bursty workload.

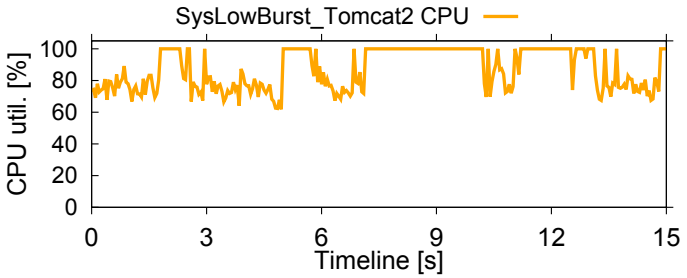
that the queue peaks in Tomcat are caused by the transient CPU bottlenecks.

The fifth step of the micro-event analysis (root cause) is the linking of transient CPU bottlenecks to the performance interferences between consolidated VMs. This is illustrated in Figure 24(c), which shows the Tomcat2 CPU utilization in SysLowBurst (reproduced from Figure 24(b)) and the MySQL request rate generated by SysHighBurst. We can see a clear overlap between the Tomcat CPU saturation periods (at 2, 5, 7-9, and 12 seconds) and the MySQL request rate jumps due to high workload bursts. The overlap indicates the transient bottlenecks in Tomcat are indeed associated with workload bursts in SysHighBurst, which created a competition for CPU in the shared node, leading to CPU saturation and queue amplification.

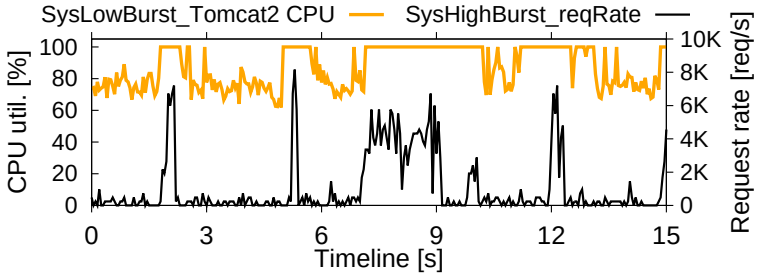
In summary, the 5 steps of micro-event analysis show the VLRT requests in Figure 23(a)



(a) Queue peaks in Apache coincide with the queue peaks in Tomcat, suggesting the pushback waves from Tomcat to Apache in SysLowBurst.

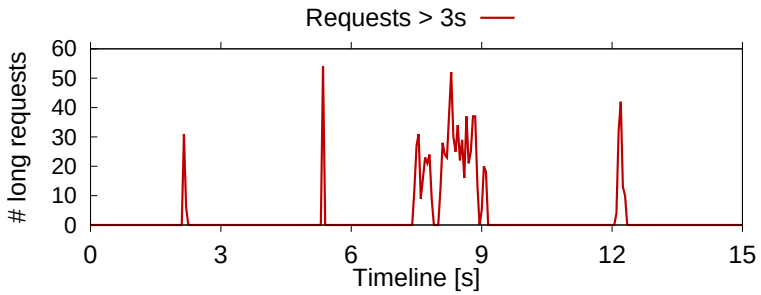


(b) Transient saturation periods of SysLowBurst-Tomcat2 CPU coincide with the same periods where Tomcat has queue peaks (see (a)).

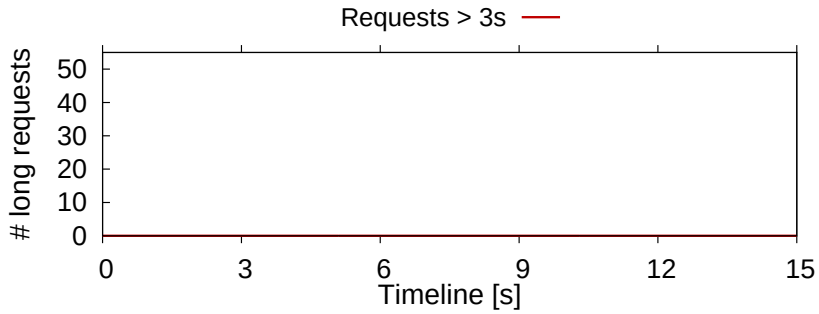


(c) The workload bursts for SysHighBurst coincide with the transient CPU saturation periods of SysLowBurst-Tomcat2, indicating severe performance interferences between consolidated VMs.

Figure 24: Transient bottlenecks caused by the interferences among consolidated VMs lead to queue peaks in *SysLowBurst-Apache*. The VM interferences is shown in (b) and (c).



(a) SysLowBurst with consolidation; the number of VLRT requests measured during a 15-second time period.



(b) SysLowBurst without consolidation; the number of VLRT requests measured during a 15-second time period.

Figure 25: SysLowBurst at workload 14000 has not VLRT requests without VM consolidation.

are due to transient bottlenecks caused by the interferences among consolidated VMs:

1. Transient events: VLRT requests are clustered within a very short period of time (4 times in Figure 23(a)).

2. Retransmitted requests: The VLRT requests correspond to periods of similar short duration, in which long request queues form in Apache server (Figure 23(b)), causing dropped packets and TCP retransmission after 3 seconds.

3. Queue amplification: The long queues in Apache are caused by push-back waves from Tomcat, where similar long queues form at the same time (Figure 24(a)).

4. Transient bottlenecks: The long queues in Tomcat (Figure 24(a)) are created by transient bottlenecks (Figure 24(b)), in which the Tomcat CPU becomes saturated for a short period of time.

5. Root cause: The transient bottlenecks are caused by the interferences among consolidated VMs (Figure 24(c)).

We note that without VM consolidation, SysLowBurst at workload 14000 has not VLRT requests as shown in Figure 25. This means that a simple way to to avoid transient bottlenecks due to VM consolidation is through VM migration and disabling the VM consolidation. This solution may not be preferred in shared cloud environment, a more sophisticated solution is one of our future research.

3.4 Modeling of Transient Bottlenecks Using "Temporary Resource Shortage"

So far we have introduced the transient bottlenecks caused by different factors from different system layers, which lead to the latency long-tail problem. As we can see that the essential problem of a transient bottleneck, regardless of the cause, is the temporary resource shortage (e.g., temporary CPU shortage) during the bottleneck period, which causes requests to queue in the bottlenecked server and potentially in the upper tiers due to the push-back waves. Thus a natural question comes after this observation: *what exactly is the relationship between the duration of a temporary resource shortage and its negative impact on the end-to-end response time variations?* This is an important question since the temporary resource shortage caused by a transient bottleneck is usually at the time scale of tens to few hundreds of milliseconds, which is apparently too short for application level measurements (often done at periods of multiple seconds) not to be taken into consideration when measuring application level performance. In this section, we discuss a simple model to quantify the negative impact of transient bottlenecks on the end-to-end response time.

3.4.1 Simuluation Setup

We use an open source queuing network simulator JMT [22] which simulates a real 3-tier experimental configuration as shown in Figure 26. To simplify the analysis we choose four representative types of requests, $T1$, $T2$, $T3$, and $T4$ to constitute the simulation workload. $T1$ requests only reaches the Apache server tier and return back to the clients. This type of requests simulate the requests for static content such as images, CSS. $T2$ requests only

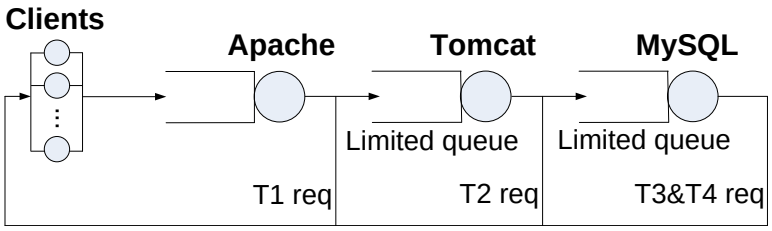


Figure 26: Simulation setup for sensitivity analysis

reach the Tomcat tier and return back. Such type of requests only need simple calculation in the application server tier. $T3$ and $T4$ requests will reach MySQL since they need dynamic content from database. To make the simulation results comparable to the real experimental results, the service time and the percentage of each type of requests are parameterized based on the real RUBBoS experimental measurements. The queue size of Apache, Tomcat, and MySQL in simulation is set to be 400, 200, 60, which are derived from the real experimental configurations to simulate the concurrency limit caused by soft resource pool size (e.g., connection/thread pool size).

3.4.2 Algorithmic Calculation of Negative Impact of Temporary Resource Shortage

The study of the negative impact is based on a simple 3-tier application model using the detailed simulator. We use the term *queued requests* to denote the negative impact of a temporary resource shortage on an n-tier application performance. This is because (1) the number of queued requests has a strong correlation with the end-to-end system response time; (2) the number of queued requests is a direct way to show the push-back waves between consecutive tiers in the system. The negative impact in each time_window can be calculated as follows:

1. $Queue\_req = Required\_work - Work\_done$
2. $Required\_work = legacy\_queue\_req + incoming\_req$
3. $Work\_done = Throughput$ * time_window

In the above definition $Queue\_req$ is the difference between $Required\_work$ and $Work\_done$. Since $Required\_work$ mainly depends on the application-level workload ($incoming\_req$), a

temporary resource shortage impacts *Queue_req* mainly by changing the *Work_done*. In fact a temporary resource shortage impacts *Work_done* of a server not only by the duration of the resource shortage, but also by the intensity of the resource shortage. For example, given the same duration of a resource shortage, the resource shortage caused by JVM GC is more severe than that caused by the delay of CPU DVFS adaption. This is because during a JVM GC period, the CPU resource is 100% utilized for cleaning garbage in memory; the *Work_done* (for request processing) is nearly zero. On the other hand, during the delay period of CPU DVFS adaption, the *Work_done* is at least half of the maximum since the CPU frequency of the slowest P8-state is about half of the fastest P0-state. So the worst case of a temporary resource shortage should be of long duration and high intensity.

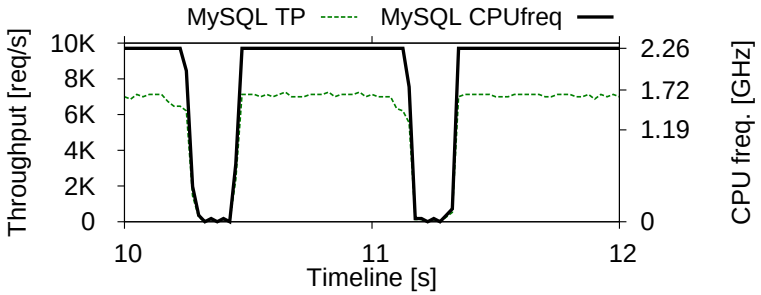
In the simulator, we calculate *Queue_req* in each time_window by the following steps:

- For each task being launched, increase the *Required_work* counter by one.
- For every task completed, increase *Work_done* counter by one.
- For every time_window, subtract the *Work_done* from *Required_work*.
- The remaining of *Required_work* is the *legacy_queue_req* for the near future.

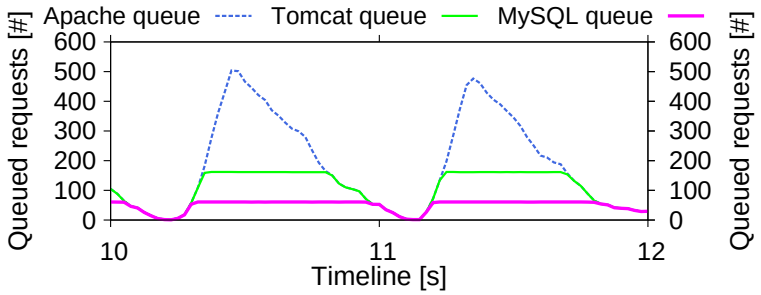
This algorithm reflects the observation that *Queue_req* in each time window is accumulated among consecutive time windows. Given this algorithm, we are able to quantify the negative impact of a temporary resource shortage on system performance, which is necessary for the sensitivity analysis in the next section.

3.4.3 Impact of the Duration of Temporary Resource Shortage

In this section we analyze the magnitude of queued requests in the entire system as a function of the duration of a resource shortage using the detailed simulator. For simplicity, we assume the resource shortage here means complete resource unavailability during the period (similar to the JVM GC case). We found that even if the duration of a resource shortage is as short as 80ms, it can cause over 500 queued requests in the front tier of the system due to push-back waves, which may far exceed the concurrency limit of a real system and cause frequent TCP retransmissions.



(a) The drop of MySQL CPU frequency indicates short-term CPU resource shortage, causing MySQL throughput drop to zero.

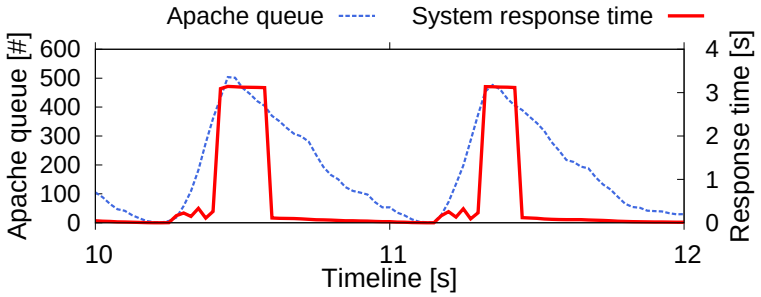


(b) Due to CPU resource shortage (Figure 27(a)), the queues in MySQL fill up during the transient bottleneck, causing the MySQL to block new requests. This "push-back" forces the requests to queue up in the upstream Tomcat and Apache.

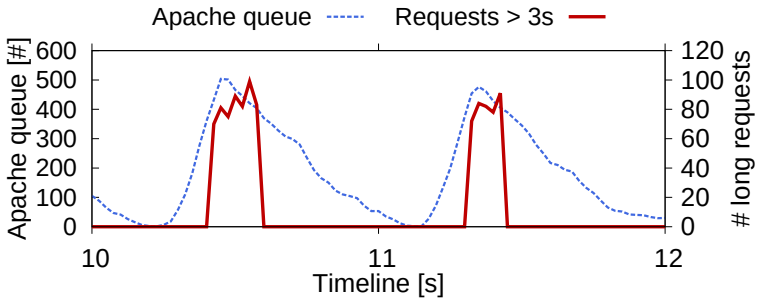
Figure 27: Simulation analysis of transient bottlenecks in MySQL causing high queue in Tomcat and Apache. The transient bottlenecks are caused by the short-term CPU resource shortage in MySQL (Figure 27(a)).

We use the detailed simulator (See Appendix 3.4.1) to study the negative impact of the duration of a temporary resource shortage on a real 3-tier RUBBoS experimental configuration as shown in Figure 26. To clearly illustrate the push-back waves, we make the temporary resource shortage appear at the bottom most MySQL tier.

Figure 27 shows the simulation analysis of transient bottlenecks caused by temporary CPU shortage in MySQL. Figure 27(a) shows that the drops of MySQL CPU frequency from the maximum to zero matches the drops of MySQL throughput, which indicates temporary CPU resource shortage (e.g., the period between time marker 10.3 and 10.4). Figure 27(b) shows that the temporary CPU shortage soon pushes requests to queue in MySQL and upper tiers to high peaks (about 500) due to the push-back waves from MySQL to Apache. High peaks of queued requests in Apache lead to peak system response time and he requests with response time longer than 3s as shown in Figure 28(a) and 28(b).



(a) Apache queuing causes the average system response time to increase beyond the duration of the transient bottlenecks.

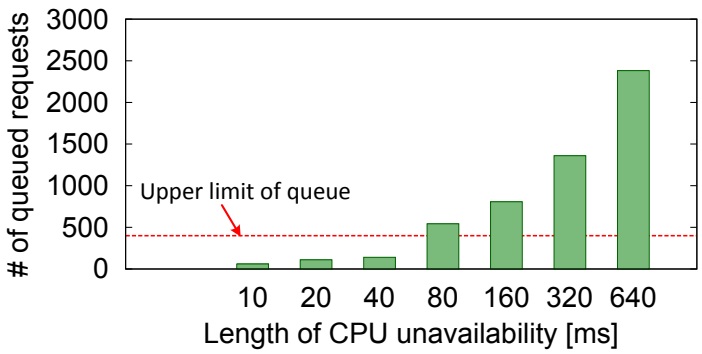


(b) Apache queue and the number of requests with long response time

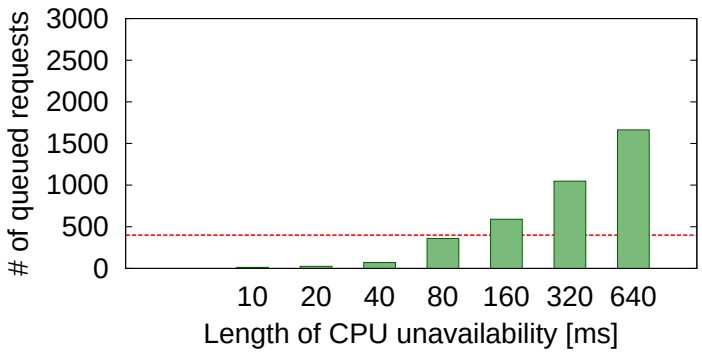
Figure 28: Illustration of high queue in Apache causing long system response time.

The simulation results above confirm our previous three case studies where temporary resource shortage caused by transient bottlenecks, regardless of the root cause, leads to large response time variations.

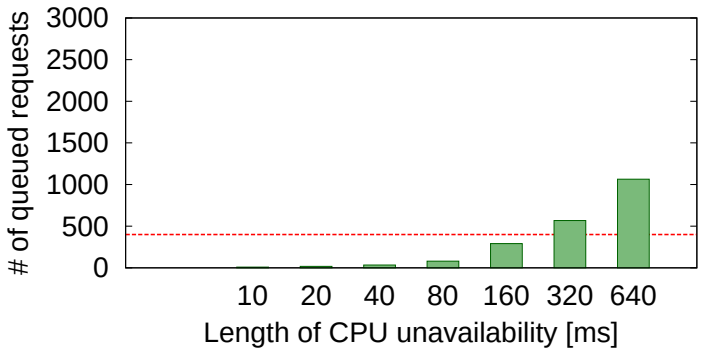
Figure 29 further shows the impact of the duration of CPU shortage on the number of queued requests in the 3-tier system at three different CPU utilization levels of MySQL. Different utilization levels of MySQL is due to the different workload intensity for MySQL, which is an important factor for the negative impact of a temporary resource shortage based on our negative impact definition in the previous subsection. All these three subfigures show that longer CPU shortage causes higher number of queued requests in the system as expected. Specially, Figure 29(a) shows even if the CPU shortage is as short as 80ms, it causes over 500 requests to queue in the system. The horizontal line in this figure shows the concurrency limit of our real experimental configuration of the 3-tier RUBBoS benchmark. Thus Figure 29(a) shows that even a short duration of CPU shortage can cause high peak of queued requests that exceeds the concurrency limit of the system, which leads to frequent



(a) Average MySQL CPU utilization at 80%.



(b) Average MySQL CPU utilization at 60%.



(c) Average MySQL CPU utilization at 40%.

Figure 29: Simulation analysis of the impact of the duration of a transient bottleneck in MySQL on the total number of queued requests in the system when the system is at different utilization levels. The red dashed line in each sub-figure indicates the concurrency limit (400) of Apache. These three sub-figures show that as the duration of a transient bottleneck becomes longer and the system utilization becomes higher, the negative impact of the transient bottleneck becomes much larger.

TCP retransmissions and requests with long response times. Figure 29(b) and 29(c) show that as the resource utilization of the system becomes lower, the negative impact of a transient resource shortage decreases due to less number of incoming requests arrives into the system.

Overall, the analysis in this section shows that in order to mitigate the negative impact of transient bottlenecks, we need to run an n-tier web application at appropriate utilization by delimiting a safe workload region for the application. How to derive the safe workload region using an analytical way given the configuration of an n-tier application is one of our future research.

3.5 Related Work

Latency has received increasing attention in the evaluation of quality of service provided by computing clouds and data centers [15, 64, 69, 78, 80]. Specifically, the long-tail latency is of particular concern for mission-critical web-facing applications [13, 14, 32, 49, 84]. On the solution side, Dean et al. [32] described their efforts to mitigate tail latency in Google's interactive applications. These bypass techniques are effective in specific applications or domains, contributing to an increasingly acute need to improve our understanding of the general causes for the VLRT requests.

Aggregated statistical analyses over fine-grained monitored data have been used to infer the appearance and causes of long-tail latency [31, 49, 80]. Li et al. [49] measure and compare the changes of latency distributions to study hardware, OS, and concurrency-model induced causes of tail latency in typical web servers executing on multi-core machines. Wang et al. [80] propose a fine-grained correlation analysis between a server's throughput and concurrent jobs in the server to infer the server's real-time performance state. Cohen [31] use a class of probabilistic models to correlate system-level metrics and threshold values with high-level performance states. Our work leverages the fine-grain data, but we go further in using micro-level timeline event analysis to link the various causes to VLRT requests.

Our work makes heavy use of data from fine-grained monitoring and profiling tools [4, 7]. Related techniques have been proposed to help detect a performance problem and identify

symptoms associated with the problem [23, 25, 49, 65, 68]. For example, Collectl [4] provides the ability to monitor a broad set of system level metrics such as CPU and I/O operations at millisecond-level granularity. Chopstix [23] continuously collects profiles of low-level OS events (e.g., scheduling, L2 cache misses, page allocation, locking) at the granularity of executables, procedures and instruction. Li et al. [49] propose a fine-grained timestamping technique to measure how much time a request spends in different parts of the server OS. We use these tools when applicable.

Techniques based on end-to-end request-flow tracing have been proposed for performance anomaly diagnosis [12, 16, 27, 36, 67, 70], but usually for more stable and longer phenomena. X-ray [16] instruments binaries as applications execute and uses dynamic information flow tracking to estimate the likelihood that a block was executed due to each potential root cause for the performance anomaly. Fay [36] provides dynamic tracing through use of run-time instrumentation and distributed aggregation within machines and across clusters for windows platform. Aguilera et al. [12] infer causal paths between component servers in a distributed system and attribute delays to specific nodes. Pip [67] detects anomalous requests by comparing request-flows from actual behaviors with developer-expected behaviors. Spectroscope [70] is similar to Pip, but Spectroscope compares request-flows between "problem" periods and "non-problem" periods for identifying anomalous requests.

3.6 Conclusion

Applying a micro-level event analysis on extensive experimental data collected from fine-grain monitoring of n-tier application benchmarks, we demonstrate that the latency long tail problem can have several causes at three system layers. Specifically, very long response time (VLRT) requests may arise from CPU DVFS control at the architecture layer (Section 3.3.2), Java garbage collection at the system software layer (Section 3.3.1), and interferences among virtual machines (VM) in VM consolidation at the VM layer (Section 3.3.3). Despite their different origins, these phenomena can be modeled and described as *transient bottlenecks*, very short periods of time (tens to hundreds of milliseconds) in which the CPU is saturated. The micro-level event analysis shows the VLRT requests are coincidental to

transient bottlenecks in various servers, which in turn amplify queuing in upstream servers, quickly leading to TCP buffer overflow and request retransmission, causing VLRT requests of several seconds.

We further build a simple model to quantify the negative impact of a transient bottleneck on system performance (Section 3.4). Our simulation results show that we can avoid/mitigate the large response time variations caused transient bottlenecks by delimiting a safe utilization level of an n-tier web application. However, finding the safe utilization level of an n-tier web application is non-trivial because it depends on many factors such as hardware/software configurations, workload characteristics, and also the system state. We believe that our study of transient bottlenecks uncovered only the "tip of iceberg" and the search for effective remedies for transient bottlenecks has only just begun.

CHAPTER IV

A GENERIC TRANSIENT BOTTLENECKS DETECTION METHOD

In Chapter 2 and 3 we showed concrete experimental evidence of the presence of the latency long-tail problem and explicitly linked transient bottlenecks as an important contributing factor to the problem. However, given no a priori knowledge of the transient bottlenecks, how to identify the location of transient bottlenecks is a significant challenge when scaling n-tier applications in computing clouds. As we have showed that transient bottlenecks arise frequently at high resource utilization and often result from transient events (e.g., JVM garbage collection) in an n-tier system and bursty workloads. Because of their short lifespan (e.g., milliseconds), these transient bottlenecks are difficult to detect using current system monitoring tools with sampling at intervals of seconds or minutes.

In this chapter, we describe a novel transient bottleneck detection method that correlates throughput (i.e., request service rate) and load (i.e., number of concurrent requests) of each server in an n-tier system at fine time granularity. Both throughput and load can be measured through passive network tracing at millisecond-level time granularity. Using correlation analysis, we can identify the transient bottlenecks at time granularities as short as 50ms. We validate our method experimentally through two case studies on transient bottlenecks caused by factors at the system software layer (e.g., JVM garbage collection) and architecture layer (e.g., Intel SpeedStep).

4.1 Introduction

Achieving both good performance and high resource utilization is an important goal for enterprise cloud environments. High utilization is essential for high return on investment for cloud providers and low sharing cost for cloud users. Good performance is essential for mission-critical applications (e.g., web-facing e-commerce applications) with Service Level Agreement (SLA) guarantees such as bounded response time. Unfortunately, achieving both objectives for mission-critical applications has remained an elusive goal. Concretely,

both practitioners and researchers have experienced wide-range response time variations in clouds during periods of high utilization. A practical consequence is that enterprise cloud environments have adopted conservative (low) average utilization (e.g., 18% in [74]).

In this paper, we describe clear experimental evidence that shows transient bottlenecks being an important contributing factor to the wide response time variations. Using extensive measurements of an n-tier benchmark (RUBBoS [8]), we demonstrate the presence of transient bottlenecks with a short lifespan on the order of tens of milliseconds. Transient bottlenecks can arise from several factors at different system layers such as Java Virtual machine garbage collection (JVM GC) at the software layer and Intel SpeedStep at the architecture layer. These factors interact with normal bursty workloads [54] from clients, often leading to transient bottlenecks that cause overall performance degradation. The discovery of these transient bottlenecks is important as they will cause wide-range response time variations and limit the overall system performance while all the system resources are less than 100% utilized. Specifically, we have found that frequent transient bottlenecks can cause a long-tail response time distribution that spans a spectrum of 2 to 3 orders of magnitude, which can lead to severe violations of strict Service Level Agreements (SLAs) required by web-facing e-commerce applications (see Section 4.2.2).

The study of transient bottlenecks has been hampered due to many transient bottlenecks being short-lived (on the order of tens of milliseconds). From Sampling Theory, these transient bottlenecks would not be detectable by normal monitoring tools that sample at time intervals measured in seconds or minutes. These monitoring tools incur very high overhead at sub-second sampling intervals (about 6% CPU utilization overhead at 100ms interval and 12% at 20ms interval). By combining fine-grained monitoring tools and a sophisticated analytical method to generate and analyze monitoring data, we are able to find and study transient bottlenecks.

The first contribution of this paper is a novel transient bottleneck detection method, which is sensitive enough to detect transient bottlenecks at millisecond level. Our method uses passive network packet tracing, which monitors the arrival and departure time of each request of each server at microsecond granularity with negligible impact on the servers.

This data supports the counting of concurrent requests and completed requests at fine time granularity (e.g., 50ms). For sufficiently short time intervals, we can use the server request completion rate as throughput, and concurrent requests as server load, to identify transient performance bottlenecks (Utilization Law [34]) at time granularity as short as 50ms (See Section 4.3).

The second contribution of the paper is a detailed study of various system factors that cause the transient bottlenecks in the system. In this paper we focus on two representative factors: one at the system software layer and the other at the architecture layer. At the system software layer, JVM garbage collections in a Java-based server happen frequently especially when the server is at high resource utilization and cause frequent transient bottlenecks for the server (see Section 4.4.1). At the architecture layer, the Intel SpeedStep technology unintentionally creates frequent transient bottlenecks due to the mismatch between the current CPU clock speed and the bursty real-time workload on the server (See Section 4.4.3).

The rest of the paper is organized as follows. Section 4.2 shows the wide-range response time variations using a concrete example. Section 4.3 introduce our transient bottleneck detection method. Section 4.4 shows two case studies of applying our method to transient bottlenecks. Section 4.5 summarizes the related work and Section 4.6 concludes the paper.

4.2 Background and Motivation

4.2.1 Experimental Setup

We adopt the RUBBoS standard n-tier benchmark, based on bulletin board applications such as Slashdot [8]. RUBBoS can be configured as a three-tier (web server, application server, and database server) or four-tier (addition of clustering middleware such as C-JDBC [26]) system. The workload consists of 24 different interactions. The benchmark includes two kinds of workload modes: browse-only and read/write mixes. We use browse-only workload in this paper.

We run the RUBBoS benchmark on our virtualized testbed. Figure 30 outlines the software components, ESXi host and virtual machine (VM) configuration, and a sample

Software Stack

Web Server	Apache 2.0.54
Application Server	Apache Tomcat 5.5.17
Cluster middleware	C-JDBC 2.0.2
Database server	MySQL 5.0.51a
Sun JDK	jdk1.5.0_07, jdk1.6.0_14
Operating system	RHEL 6.2 (kernel 2.6.32)
Hypervisor	VMware ESXi v5.0
System monitor	esxtop 5.0, Sysstat 10.0.0

(a) Software setup

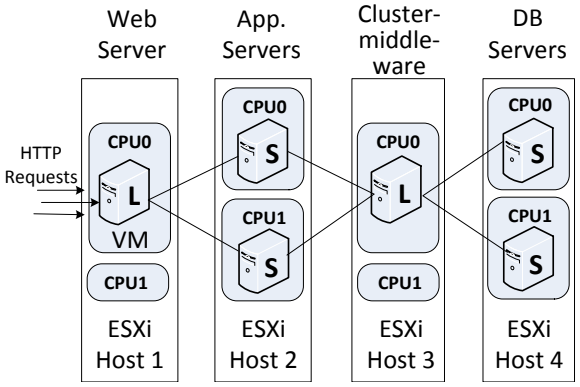
ESXi Host Configuration

Model	Dell Power Edge T410
CPU	2* Intel Xeon E5607, 2.26GHz Quad-Core
Memory	16GB
Storage	7200rpm SATA local disk

VM Configuration

Type	# vCPU	CPU limit	CPU shares	vRAM	vDisk
Large (L)	2	4.52GHz	Normal	2GB	20GB
Small (S)	1	2.26GHz	Normal	2GB	20GB

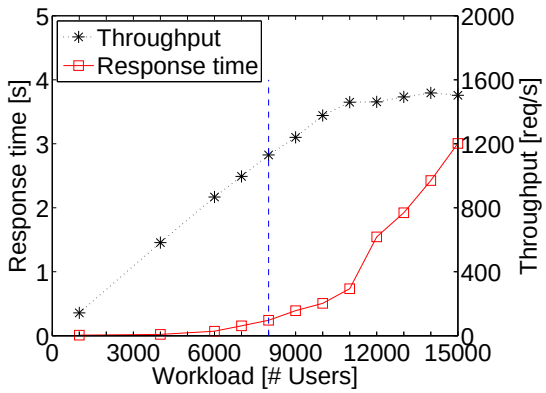
(b) ESXi host and VM setup



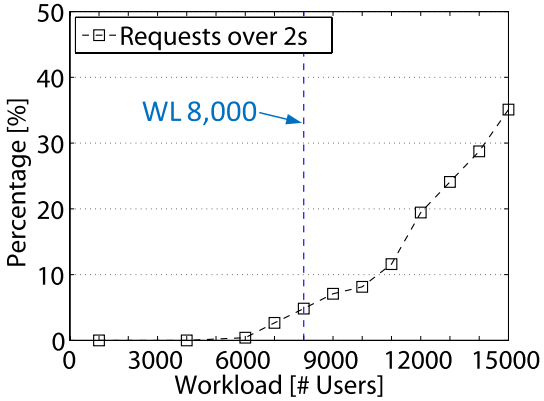
(c) 1L/2S/1L/2S sample topology

Figure 30: Details of the experimental setup.

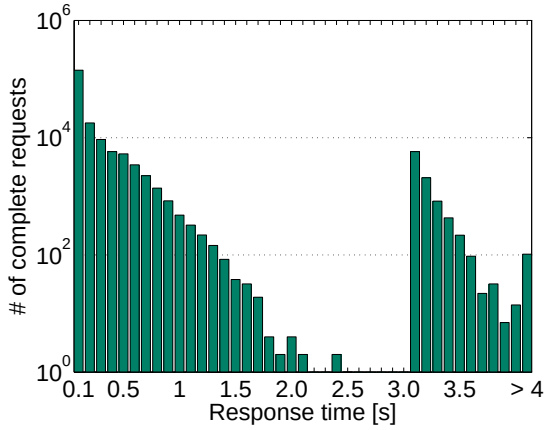
topology used in the experiments. We use a four-digit notation $\#W/\#A/\#C/\#D$ to denote the number of web servers, application servers, clustering middleware servers, and database servers. Each server runs on top of one VM. We have two types of VMs: "L" and "S", each of which represents a different size of processing power. Figure 30(c) shows a sample 1L/2S/1L/2S topology. The VMs from the same tier of the application run in the same ESXi host. Each VM from the same tier is pinned to separate CPU cores to minimize the interference between VMs. Hardware resource utilization measurements (e.g., CPU) are taken during the runtime period using Sysstat at one second granularity and VMware esxtop at two second granularity.



(a) Average end-to-end response time and throughput at each workload



(b) Percentage of requests with response time over two seconds at each workload



(c) Long-tail and bi-modal end-to-end response time distribution at WL 8,000

Figure 31: A case where the system response time shows wide-range variation far before the system reaches the maximum throughput. Figure 31(c) shows the long-tail and bi-modal end-to-end response time distribution at WL 8,000, which indicates the unstable system performance.

4.2.2 Why Are Transient Bottlenecks a Problem?

We use an example where the response time of an n-tier system presents wide-range variations while the system is far from saturation. The example was derived from a three-minute experiment of RUBBoS running on a four-tier configuration (1L/2S/1L/2S, see Figure 30(c)).

Figure 31(a) shows the system throughput increases linearly from a workload of 1,000 concurrent users to 11,000, but after 11,000, the throughput becomes flat and the average

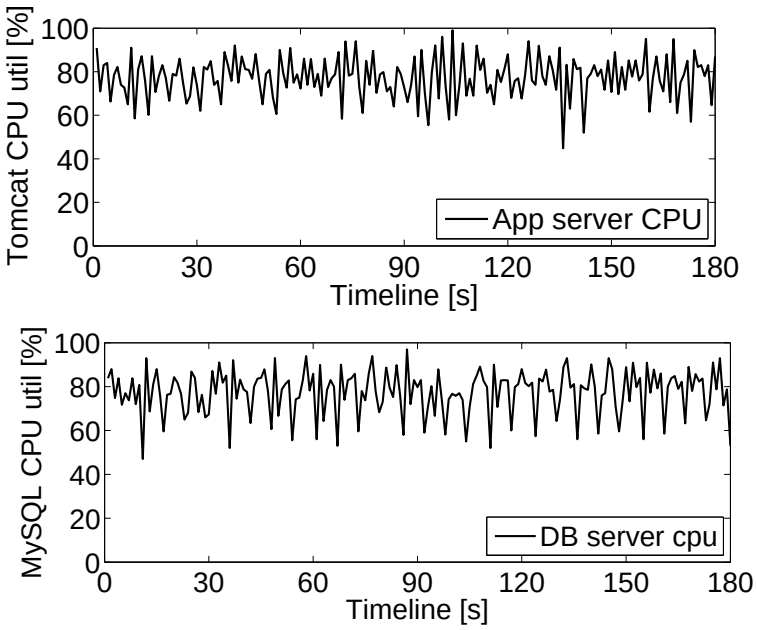


Figure 32: Tomcat and MySQL CPU utilization at WL 8,000; the average is 79.9% and 78.1% respectively.

response time increases dramatically. The interesting observation is that before the throughput reaches the maximum, for example, from WL 6,000 to 11,000, the average response time already starts increasing. In particular, Figure 31(b) shows that the percentage of requests with response time over 2s starts increasing after WL 6,000, which means that the system performance starts deteriorating far before the system reaches the maximum throughput. Figure 31(c) further shows the response time distribution of the system at WL 8,000, which presents a clear long-tail and bi-modal distribution. In real business situations, there are often cases when web-facing applications have strict service level agreements (SLAs) in terms of end-to-end response time; for example, experiments at Amazon show that every 100ms increase in the page load decreases sales by 1% [48]. In such cases, wide-range variations in response time can lead to severe SLA violations.

In order to diagnose the causes for the wide-range response time variations, we measured the utilization of various resources in each component server of the system. Since the browse-only workload of RUBBoS is CPU intensive, we show the timeline graphs (with one second granularity) of CPU utilization in Figure 32. During the execution of the WL 8,000, both Tomcat and MySQL show less than full CPU utilization, with an average of

Table 7: Average resource utilization in each tier at WL 8,000. Except Tomcat and MySQL CPU, the other system resources are far from saturation.

Server/Resource	CPU util. (%)	Disk I/O (%)	Network receive/send (MB/s)
Apache	34.6	0.1	14.3/24.1
Tomcat	**79.9**	0.0	3.8/6.5
CJDBC	26.7	0.1	6.3/7.9
MySQL	**78.1**	0.1	0.5/2.8

79.9% (Tomcat) and 78.1% (MySQL). We also summarize the average usage of other main hardware resources of each server in Table 7. This table shows that except for Tomcat and MySQL CPU, the other system resources are far from saturation.

This example shows that monitoring hardware resource utilization at one second granularity is insufficient at identifying the cause of wide-range response time variations, since there is no single saturated resource. Later in Section 4.4.3 we explain that the problem is due to the frequent transient bottlenecks unintentionally caused by Intel SpeedStep technology in MySQL. SpeedStep is designed to adjust CPU clock speed to meet instantaneous performance needs while minimizing the power consumption of CPUs; however, the Dell's BIOS-level SpeedStep control algorithm is unable to adjust the CPU clock speed quickly enough to match the bursty real-time workload; the mismatch between CPU clock speed and real-time workload causes frequent transient bottlenecks in MySQL and leads to wide-range variations of system response time [1].

4.2.3 Trace Monitoring Tool

The previous example shows the necessity of detecting transient bottlenecks in the system. Our approach is based on passive network tracing, which can mitigate the monitoring overhead while achieve high precision of detecting transient bottlenecks in the system. In this section, we introduce our monitoring tool, which we use in our transient bottleneck detection method presented in the next section.

[1] Transient bottlenecks cause instantaneous high concurrency in an n-tier system; once the concurrency exceeds the thread limit in the web tier of the system, new incoming requests will encounter TCP retransmissions, which cause over 3s response times [78].

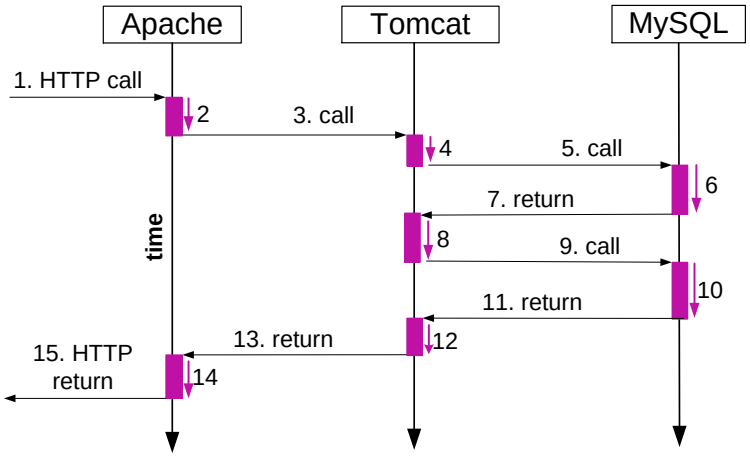


Figure 33: Illustration of a transaction execution trace captured by SysViz.

We use Fujitsu SysViz [2] to monitor the trace of transaction executions in our experiments. Figure 33 shows an example of such a trace (numbered arrows) of a client transaction execution in a three-tier system. A client transaction services an entire web page requested by a client and may consist of multiple interactions between different tiers. SysViz is able to reconstruct the entire trace of each transaction executed in the system based on the interaction messages (odd-numbered arrows) collected through network taps or network switches which support passive network tracing. Since the timestamp of each interaction message is recorded on one dedicated SysViz machine and independent of clock errors caused by limited accuracy of NTP, the intra-node delay (small boxes with even-numbered arrows) of every request in any server in the system can be precisely recorded.

SysViz processing is based on four steps. (1) Collect all IP packets going through the n-tier system by using port mirroring function of network switches and forward them to a dedicated SysViz server. (2) The SysViz server translates the IP packets to protocol messages (e.g., HTTP and AJP) exchanged between tiers. (3) The SysViz server extracts some identification data from each protocol message (e.g., URL of an HTTP request with some CGI parameters, which are used to parameterize the SQL query of a DB request) in order to prepare the transaction trace reconstruction. (4) This step has two different modes: the training mode and the analysis mode. In the training mode, SysViz learns transaction models of the n-tier application by applying machine learning techniques on the collected

protocol messages. In the analysis mode, SysViz reconstructs the trace of each transaction of the n-tier application using the meta-information of each protocol message with precise timestamps, the learned transaction models, and a set of transaction matching algorithms.

SysViz requires no modification on application source code and has a negligible performance impact on the target n-tier application. We note that since the timestamps of all messages are assigned by one dedicated SysViz server, the precision of the derived processing time of each request in any tier in the system is close to microsecond level. Thus, the influence of clock errors between machines caused by limited accuracy of NTP can be removed.

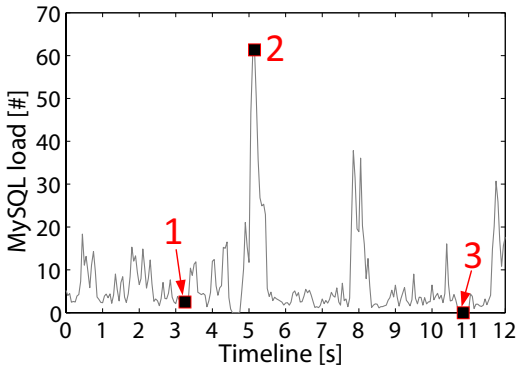
In fact the end-to-end transaction tracing has been studied for many years and there are mainly two classes of implementations: *annotation-based* and *black box*. Most annotation-based implementations [19] [27] [37] [72] rely on applications or middleware to explicitly associate each interaction message with a global identifier that stitches the messages within a transaction together. Black-box solutions [12] [18] assume there is no additional information other than the interaction messages, and use statistical regression analysis to reconstruct each transaction execution trace. SysViz belongs to the *black-box* class. Experiments in our environment shows that SysViz is able to achieve more than 99% accuracy of transaction trace reconstruction for a 4-tier application even when the application is under a high concurrent workload.

End-to-end transaction tracing in distributed systems has passed the research stage. Research continues on how to best use the information provided by such tracing to diagnose performance issues in the system.

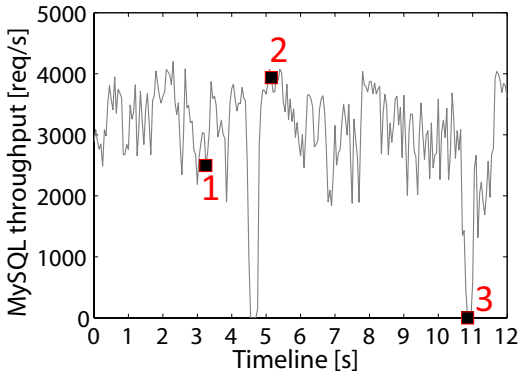
4.3 Fine-Grained Load/Throughput Analysis

In this section, we first briefly show how our method detects transient bottlenecks in an n-tier system using a simple example. The details of each part of our method are in the following subsections.

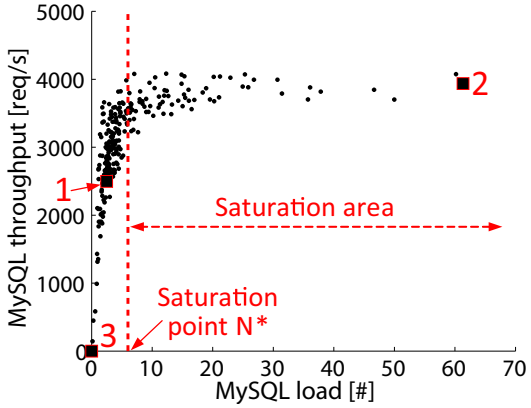
Since a bottleneck in an n-tier system is the place where requests start to congest in the system, a key point of detecting transient bottlenecks is to find component servers that



(a) MySQL load measured at every 50ms time interval in a 12-second time period. Frequent high peaks suggest that MySQL presents short-term congestions from time to time.



(b) MySQL throughput measured at every 50ms time interval in the same 12-second time period as in Figure 34(a).



(c) MySQL load vs. MySQL throughput in the same 12-second time period as in Figure 34(a) and 34(b); MySQL is temporarily congested once the load exceeds $N^{\star}$.

Figure 34: Performance analysis of MySQL using fine-grained load and throughput at WL 7,000. Figure 34(a) and 34(b) show the MySQL load and throughput measured at the every 50ms time interval. Figure 34(c) is derived from 34(a) and 34(b); each point in Figure 34(c) represents the MySQL load and throughput measured at the same 50ms time interval in the 12-second experimental time period.

frequently present short-term congestions. To achieve this goal, the first step of our method is to measure a server's load and throughput in continuous fine-grained time intervals. The throughput of a server can be calculated by counting the number of completed requests in the server in a fixed time interval, which can be 50ms, 100ms, or 1s. Load is the average number of concurrent requests over the same time interval [2]. Figure 34(a) and 34(b) shows the MySQL load and throughput measured using a 50ms time interval over a 12-second time period for the 1L/2S/1L/2S configuration case at WL 7,000 (See the case in Figure 31). These two figures show that both the MySQL load and throughput fluctuate significantly, which indicates that MySQL frequently presents short-term congestions.

To diagnose in which time intervals a server presents short-term congestion, we need to correlate the server's load and throughput as shown in Figure 34(c). This figure is derived from Figure 34(a) and Figure 34(b); each point in Figure 34(c) represents the MySQL load and throughput measured at the same 50ms time interval during the 12-second experimental time period (i.e., in total 240 points). This figure shows a clear trend of load/throughput correlation (*main sequence curve*), which is consistent with Denning et al.'s [34] operational analysis result for the relationship between a server's load and throughput. Specifically, a server's throughput increases as the load on the server increases until it reaches the *maximum throughput* TP_{max}, which is determined by the average demand for the bottleneck resource per job according to the Utilization Law. The *congestion point* $N^{\star}$ is the minimum load beyond which the server starts to congest.

Once $N^{\star}$ is determined, we can judge in which time intervals the MySQL tier is congested based on the measured load. For example, Figure 34(c) highlights three points labeled 1, 2, and 3, each of which represents the load/throughput in a time interval that can match back to Figure 34(a) and 34(b). Point 2 shows that the MySQL tier is congested in the corresponding time interval because the load far exceeds $N^{\star}$. Point 3 shows that MySQL is not congested due to the zero load. Point 1 also shows that the MySQL tier is not congested because the corresponding load is less than $N^{\star}$ though it generates high throughput.

[2] Given the precise arrival and departure timestamps of each request for a server monitored through passive network tracing, the load and throughput of the server can be calculated at any given time interval, more details are in Section 4.3.1 and 4.3.2

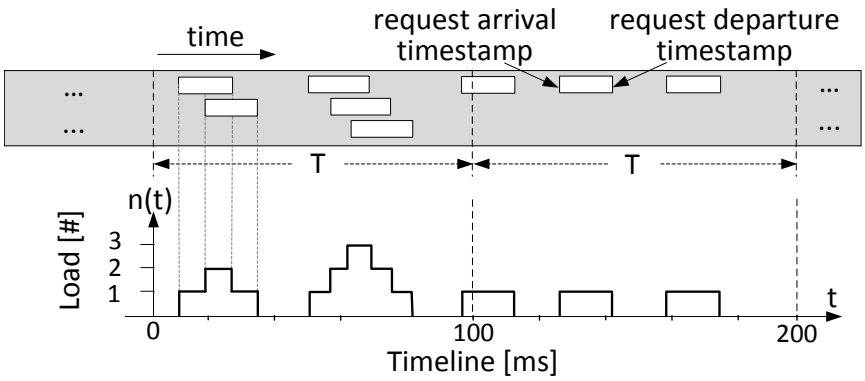


Figure 35: Load calculation for a server based on the arrival/departure timestamps of requests for the server

After we apply the above analysis to each component server of an n-tier system, we can detect which servers have encountered frequent transient bottlenecks and cause the wide-range response time variations of the system.

4.3.1 Load Calculation

For each server, our direct observables are the arriving (input) requests and departing (output) responses with timestamps generated at microsecond ticks. At each tick, we know how many requests have arrived, but not yet departed. This is the number of concurrent requests being processed by the server. We define the server load as the average number of concurrent requests over a time interval.

Figure 35 shows an example of load calculation for a server in two consecutive 100ms time intervals. The upper part of this figure shows the arrival/departure timestamps of the requests received by the server, which are collected through passive network tracing. Due to the multi-threaded architecture, requests received by a server can be processed concurrently as shown by the interleaved arrival/departure timestamps of different requests. The bottom part of this figure shows the number of concurrent requests being processed by the server at each moment; thus the average in each time interval can be calculated and used as the server load over the time interval.

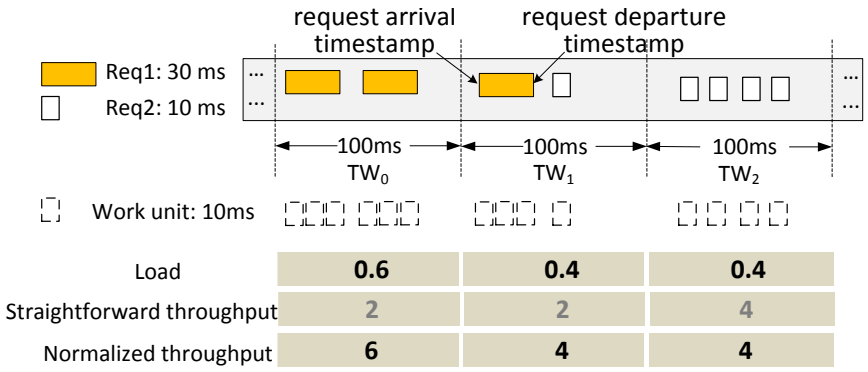


Figure 36: Load/throughput calculation with mix-class workload

4.3.2 Throughput Calculation

A straightforward approach to calculate throughput of a server in each time interval is to count the number of finished requests during each time interval. This approach is reasonable if a server processes only one class of requests because the same class of requests can be assumed to have a similar amount of demand for the bottleneck resource of the server. Thus, the throughput calculated in each time interval is comparable.

In typical applications including RUBBoS, the workload on a server is mixed with multiple classes of requests each having a different demand for the bottleneck resource of the server. As the time interval length decreases (e.g. 50ms), the request-mix distribution among time intervals becomes significantly different. Thus throughput values calculated (using the straightforward way) in different time intervals are not directly comparable because the requests that comprise the throughput may have different demands for the bottleneck resource.

To calculate the throughput of a server under a mix-class workload, we apply a throughput normalization technique which transforms different classes of completed requests into a certain number of comparable work units. [3]. We define a work unit as the greatest common divisor among the service times from different classes of requests. Requests with a longer service time can transform into a greater number of work units while those with shorter

[3]For mix-class workload, we assume the demand for the bottleneck resource of a server is proportional to the service time of a request. This assumption is reasonable if a mix-class workload is one specific resource intensive in a server (e.g., CPU). Then the service time can be approximated as CPU time.

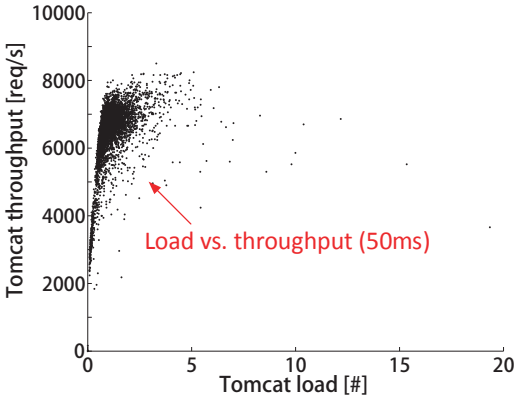
service times only transform into a smaller number. Since the normalized throughput in each time interval only takes into account the transformed work units, throughputs from different time intervals become comparable. This throughput normalization technique is motivated by the request canonicalization and clustering as introduced in Barham et al.'s Magpie [19].

Figure 36 shows an example of the load and throughput calculation under a mix with two classes of requests: Req_1 and Req_2 with service time 30ms and 10ms respectively. The time interval length is 100ms. We set the work unit size as 10ms, so then Req_1 transforms into 3 work units and Req_2 transforms into 1 work unit. Thus, the server processes 6 work units in TW_0 and 4 in both TW_1 and TW_2. We can see that in these three time intervals the normalized throughput has a strong positive correlation with the load, which means the server is not saturated based on Utilization Law. On the other hand, the number of completed requests (the straightforward throughput) has no correlation with the load in this case.

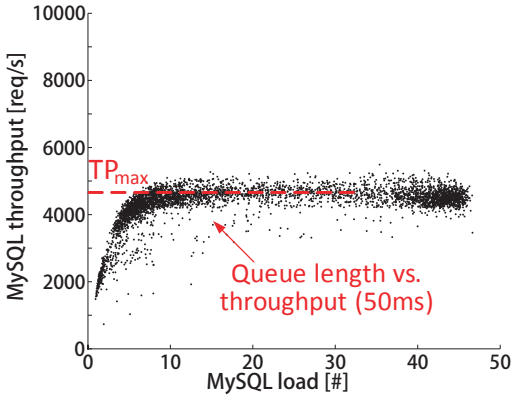
Service time approximation: The service time approximation for each class of requests is obtained using passive network tracing. Figure 33 shows the intra-node delay (small boxes in the figure) of each individual request in each server, which can be treated as the service time if there is no queuing effect. Thus, service time approximation for each class of requests can be conducted online when the production system is under low workload in order to mask out the queuing effects inside a server [76]. Since the service time of each class of requests may drift over time (e.g., due to changes in the data selectivity) in real applications, such service time approximations have to be recomputed accordingly.

4.3.2.1 Throughput Normalization Validation

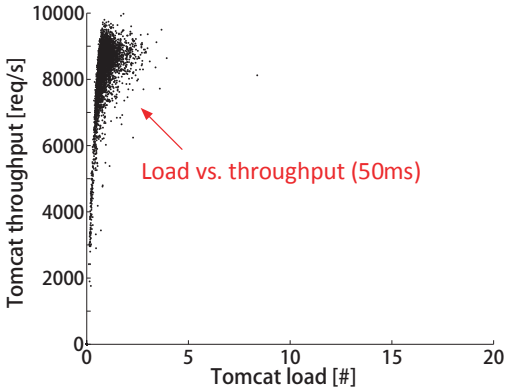
Here we show our throughput/load correlation results for two types of single-class workloads (*browseStoriesBC* and *viewComments*) and one mix-class workload (the original RUBBoS Browse-only workload; mixed with **eight** single-class workloads) under 1M/2M/1M configuration (one Apache server, two Tomcat servers, and one MySQL server). Results for the single-class workload are the baseline while the results for mix-class workload are used to



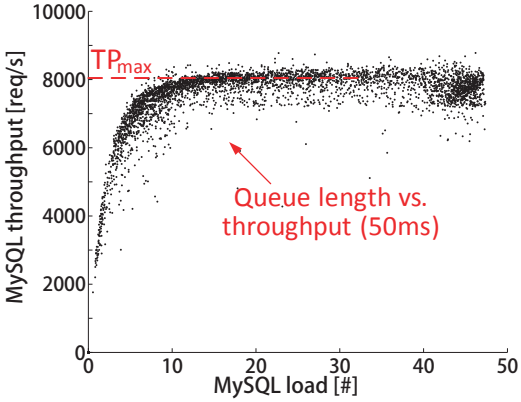
(a) Single-class (BrowseStoriesBC), Tomcat tier in WL 15400



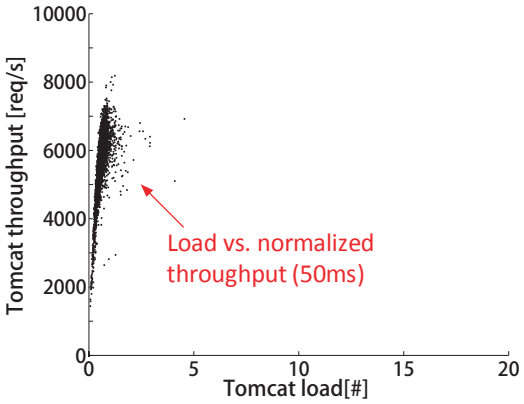
(b) Single-class (BrowseStoriesBC), MySQL tier in WL 15400



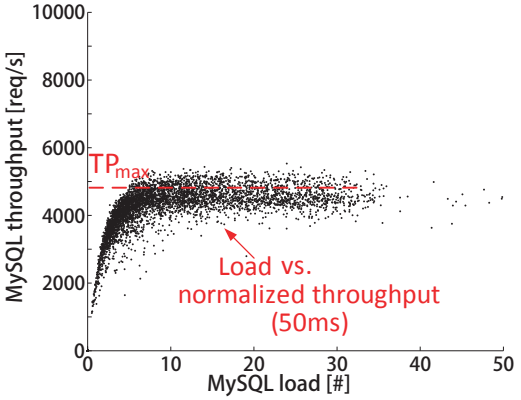
(c) Single-class (ViewComments), Tomcat tier in WL 5400



(d) Single-class (ViewComments), MySQL tier in WL 5400



(e) Mix-class (eight transactions), Tomcat tier in WL 5600



(f) Mix-class (eight transactions), MySQL tier in WL 5600

Figure 37: Load/throughput(50ms) correlation analysis for single-class and mix-class (original RUBBoS Browse-only workload, including eight classes of transactions) workload under 1S/2S/1S configuration.

validate the throughput normalization technique.

Figure 37(a) and 37(b) show the throughput/load correlation results of the Tomcat tier and the MySQL tier for the single-class workload *browseStoriesBC* under workload 15,400 while Figure 37(c) and 37(d) for the single-class workload *viewComments* under workload 5,400. We choose such specific workloads in order to show the transient saturation frequently exhibited by the MySQL server. These figures show that the load and the throughput(50ms) of both Tomcat and MySQL for a single-class workload clearly follow the expected asymptotic saturation curve. We note that MySQL achieves different maximum throughputs TP_{max} under different types of single-class workloads. This is because the service times for different classes of requests in MySQL are different, which leads to different demand for the bottleneck resource (e.g., CPU).

Figure 37(e) and 37(f) show the correlation between the load and the normalized throughput(50ms) of Tomcat and MySQL under mix-class workload. These two figures also show these two metrics for both Tomcat and MySQL clearly follows the asymptotic saturation curve, which means the throughput normalization technique is able to capture the difference among the mix-class workloads.

4.3.3 Congestion Point Determination

In our method $N^\star$ is used to classify a server's performance state in each time interval; however, the $N^\star$ of a server is not known a priori because the value depends on many factors such as the server's hardware/software configuration and also the workload characteristics [81].

In practice we use a simple statistical intervention analysis [52] to approximate $N^\star$, where the main idea of this analysis is to find the minimum load ($N^\star$) beyond which the increments of throughput becomes negligible with further increment of load. Suppose the load in a server varies between $[N_{min}, N_{max}]$; then we divide $[N_{min}, N_{max}]$ into k even intervals (e.g., $k = 100$) and calculate the average throughput in each load interval based on the load/throughput samples we collected during the experimental period. Each load interval and the corresponding average throughput is recorded as $\{\langle\overline{ld}_1, \overline{tp}_1\rangle, \langle\overline{ld}_2, \overline{tp}_2\rangle, ..., \langle\overline{ld}_k, \overline{tp}_k\rangle\}$, where $\overline{ld}_1 < \overline{ld}_2 < ... < \overline{ld}_k$. Then the slope δ_i between every two consecutive load intervals

can be calculated as Equation 1:

$$\delta_i = \begin{cases} \overline{tp}_1 / \ \overline{ld}_1 & : i = 1 \\ \dfrac{\overline{tp}_i - \overline{tp}_{i-1}}{\overline{ld}_i - \overline{ld}_{i-1}} & : 1 < i \leq k \end{cases} \quad (1)$$

$$tol \leq \overline{\delta} - t_{(0.95,n_0-1)} * s.d.\{\delta\} \quad (2)$$

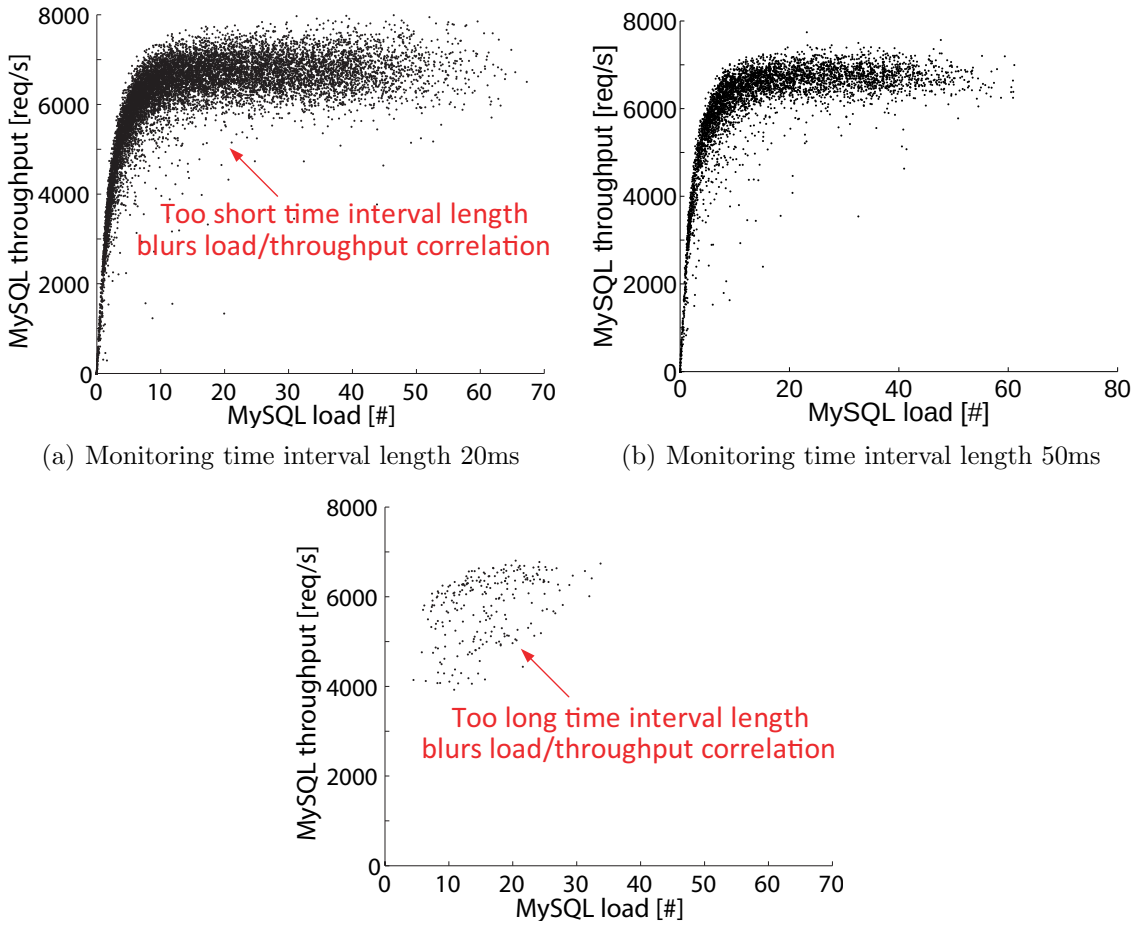
δ_i should be nearly constant (e.g., δ_0) when the server is not saturated and starts to lose stability once the load exceeds $N^\star$. The right side of Equation 2 shows a simple heuristic approximation for the lower bound of a ninety percent confidence interval of the sequence $\{\delta_1, \delta_2, ..., \delta_{n_0}\}$ [4], where $1 < n_0 \leq k$. We approximate $N^\star$ as $\overline{ld}_{n_0}$ when the lower bound of the variation of the sequence $\{\delta_1, \delta_2, ..., \delta_{n_0}\}$ is below the pre-defined threshold *tol* (e.g., $0.2\delta_0$).

4.3.4 Impact of Monitoring Time Interval Length

Both too short and too long a time interval length have side-effects in detecting transient bottlenecks of a server. Though a short time interval length can better capture the transient variation of the load of a server, it decreases the precision of the throughput calculation due to factors such as requests with a lifespan crossing consecutive time intervals or the errors caused by throughput normalization. For example, the service time even for the same class of requests varies in real applications (e.g., data selectivity changes). The average service time for the same class of requests may not be representative during throughput normalization due to too few requests completed in a small time interval. On the other hand, though a longer time interval length can average out the service time variation for the same class of requests, it may lose the ability to capture the short-term congestions of a server.

Figure 38(a), 38(b), and 38(c) show the load/throughput correlation results of MySQL

[4] $t_{(0.95,n_0-1)}$ is the coefficient for a 90 percent confidence interval when a variable follows a t-distribution; $\overline{\delta} = \frac{1}{n_0}\sum_{i=1}^{n_0}\delta_i$ and $s.d.\{\delta\} = \sqrt{\sum_{i=1}^{n_0}(\delta_i - \overline{\delta})^2}$, which are the mean and the standard deviation of the sequence $\{\delta_1, \delta_2, ..., \delta_{n_0}\}$, respectively.



(a) Monitoring time interval length 20ms

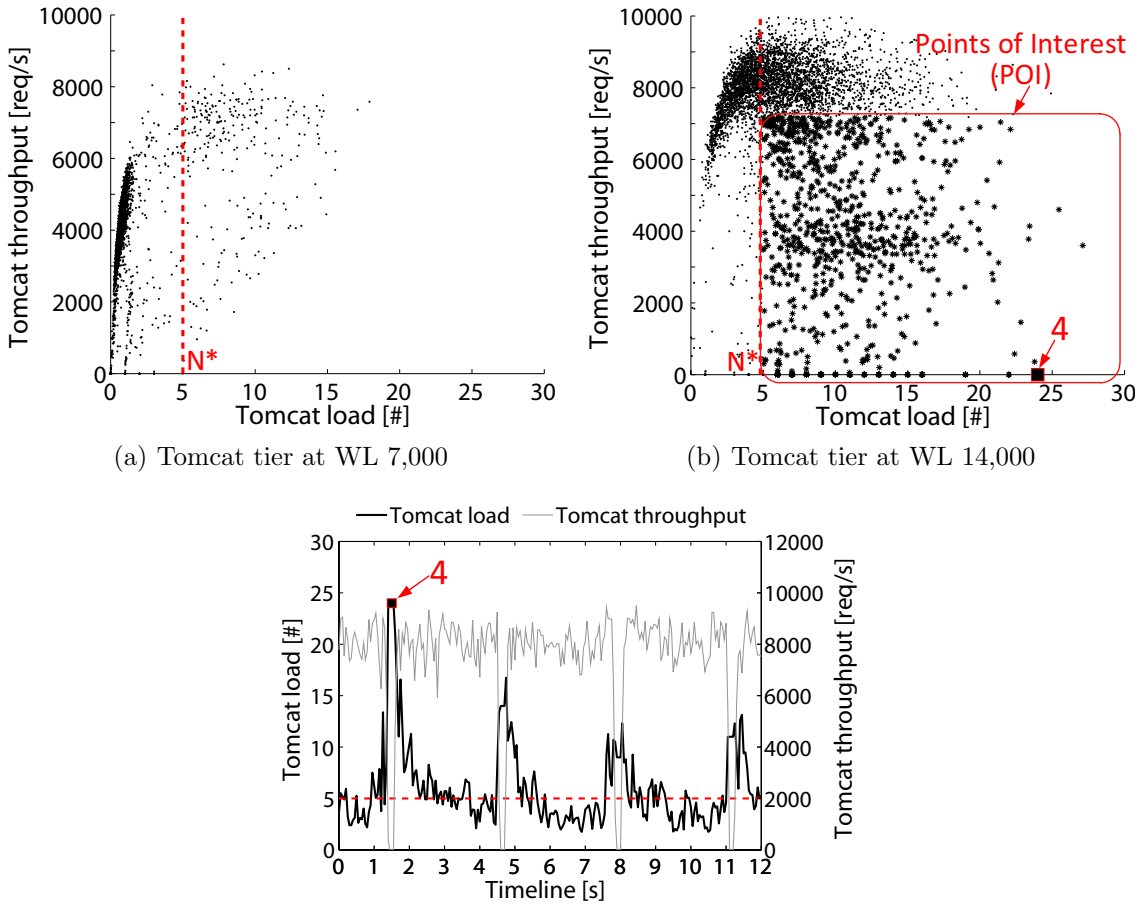
(b) Monitoring time interval length 50ms

(c) Monitoring time interval length 1s

Figure 38: The impact of time interval length on load/throughput correlation analysis for MySQL at WL 14,000. Subfigure (a) (b), and (c) are derived from the same 3-minute experimental data; thus there are 9,000 points with 20ms time interval, 3,600 points with 50ms time interval, and 180 points with 1s time interval.

at workload 14,000 with 20ms, 50ms, and 1s time interval length, respectively. Comparing these three figures we can see that too long a time interval length cannot capture the load/throughput variations, thus losing the ability to detect transient bottlenecks (Figure 38(c)); too short a time interval length blurs the shape of the expected main sequence curve due to the increased errors of normalized throughput (Figure 38(a)).

Note a proper time interval length for a server is workload dependent (e.g., depends on the service time variation of each class of requests for the server). In general a proper length should be small enough to capture the short-term congestions of a server. In the evaluation section we choose the time interval length to be 50ms. An automatic way to



(a) Tomcat tier at WL 7,000

(b) Tomcat tier at WL 14,000

(c) Tomcat tier at WL 14,000

Figure 39: Fine-grained load/throughput(50ms) analysis for Tomcat as workload increases. Subfigure 39(b) is derived from Subfigure 39(c), but with 3-minute experimental data. Subfigure 39(b) shows that Tomcat frequently presents short-term congestion at WL 14,000.

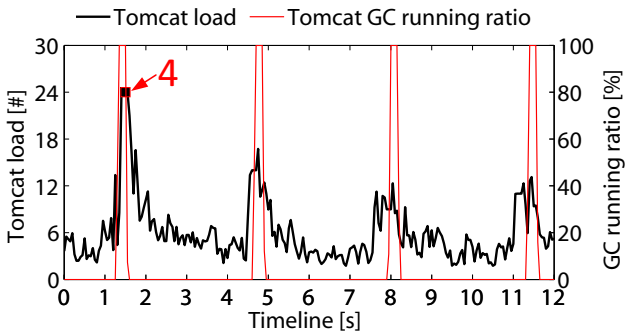
choose a proper time interval length is part of our future research.

4.4 Evaluation

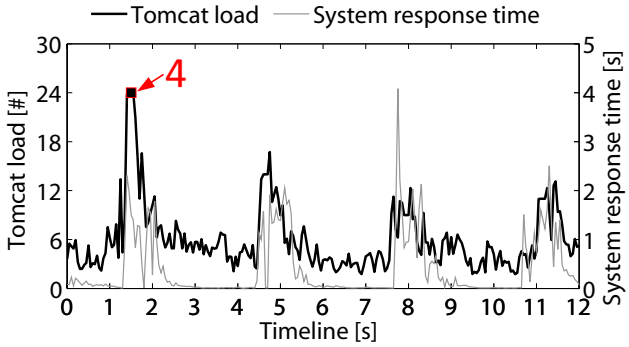
In this section we show two case studies of applying our method to detect transient bottlenecks caused by factors at different levels (e.g., JVM GC at software level and Intel SpeedStep at architecture level). For each case we also show a solution to resolve the transient bottlenecks in the system.

4.4.1 Transient Bottlenecks Caused by JVM GC

The first case is the transient bottlenecks caused by frequent JVM GCs in Tomcat. In the experiments of this subsection, we use JDK 1.5 in Tomcat which has a synchronous garbage



(a) Tomcat load vs. Tomcat GC running ratio at WL 14,000; high GC running ratio causes requests to congest in Tomcat.



(b) Tomcat load and system response time in the same time period as in (a); long queue in Tomcat leads to high peak in response time.

Figure 40: Fine-grained analysis for the large response time fluctuations of the system at WL 14,000. Figure 40(a) shows that frequent JVM GCs cause transient bottlenecks (long queue) in Tomcat, which lead to large response time fluctuations as shown in Figure 40(b).

collector; the inefficiency of this garbage collector frequently causes transient bottlenecks in Tomcat and results in significant fluctuations of system response time as we will show in Figure 41(c).

Figure 39 shows the fine-grained load/throughput (50ms) analysis for Tomcat at WL 7,000 and 14,000 with the hardware configuration 1L/2S/1L/2S. Figure 39(a) shows that Tomcat is not bottlenecked in most of the time intervals at WL 7,000 since only a few points are right after $N^\star$ derived from Figure 39(b). The interesting figure is Figure 39(b), which shows that at WL 14,000 Tomcat frequently presents transient bottlenecks. In particular, this figure shows there are many points when Tomcat has a high load but low or even zero throughput (POI inside the rectangular area), which contradicts our expectation of the

main sequence curve followed by a server's load and throughput.

To illustrate when these POIs happen, Figure 39(c) shows the fine-grained timeline analysis of Tomcat load and throughput in a 10s experimental period at WL 14,000. This figure clearly shows in some time intervals the Tomcat load is high (e.g., the point labeled 4) but the corresponding throughput is zero, which means that many requests are congested in Tomcat but there are no output responses (throughput). In such time intervals, the load/throughput pairs fall into the POI area as shown in Figure 39(b).

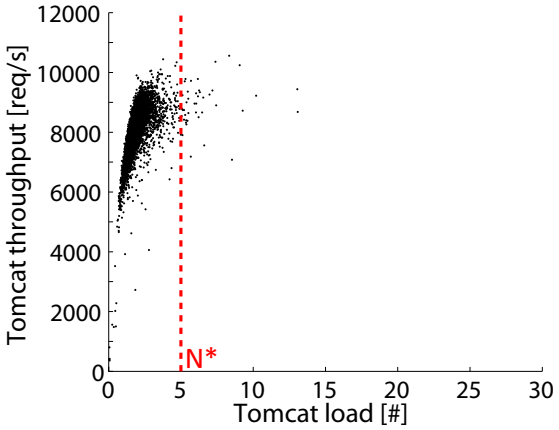
Our further analysis shows that the POIs are caused by JVM GCs that frequently stop Tomcat. In this set of experiments, the JVM in Tomcat (JDK 1.5) uses a synchronous garbage collector; it waits during the GC period and only starts processing requests after the GC is finished. To confirm that JVM GCs cause the frequent transient bottlenecks in Tomcat, Figure 40(a) shows the timeline graph which correlates the Java GC running ratio [5] with the Tomcat load. This figure shows that the occurrence of Tomcat JVM GCs have a strong positive correlation with the high peaks of load.

Figure 40(b) shows the correlation between the Tomcat load and the system response time over the same 12-second time period as in Figure 40(a). This figure shows that these two metrics positively correlate with each other, which suggests that the short-term congestions (high load) in Tomcat cause the high peaks of system response time. Figure 40(a) and 40(b) together show that frequent JVM GCs in Tomcat causes frequent short-term congestions in Tomcat, which in turn cause the significant variations on system response time.

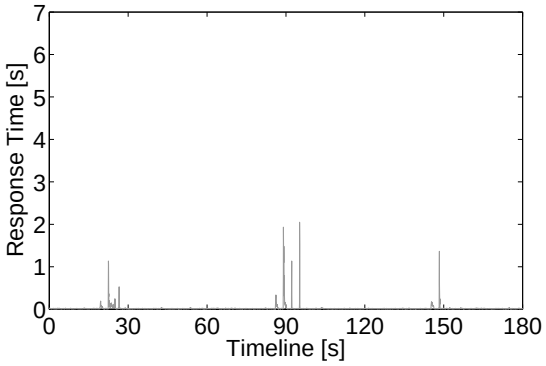
4.4.2 Solution: Upgrade JDK Version in Tomcat

Once we detect the frequent transient bottlenecks in Tomcat, we can resolve such bottlenecks by simply scaling-out/up the Tomcat tier since low utilization of Tomcat can reduce the negative impact of JVM GC [78]. Here we illustrate a more economical way to solve the problem by just upgrading the Tomcat JDK version from 1.5 to 1.6, which has more efficient

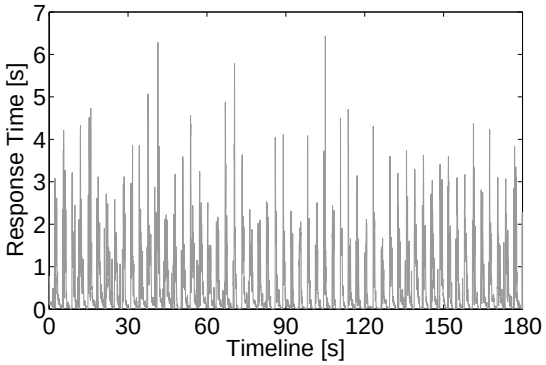
[5] Java GC running ratio means the percentage of time spent on Java GC in each monitoring time interval. JVM provides a logging function which records the starting/ending timestamp of every GC activity.



(a) Tomcat tier at WL 14,000 (JDK 1.6 in Tocmat)



(b) System response time averaged in every 50ms (JDK 1.6 in Tocmat)



(c) System response time averaged in every 50ms (JDK 1.5 in Tocmat)

Figure 41: Resolving transient bottlenecks by upgrading Tomcat JDK version from 1.5 to 1.6. Figure 41(a) shows that the frequent transient bottlenecks in Tomcat as shown in Figure 39(b) are resolved. Thus, comparing Figure 41(b) and 41(c), the system response time presents much less fluctuations.

garbage collectors[6]. The experimental configurations are kept the same as before except the Tomcat JDK version.

Figure 41(a) shows the fine-grained load/throughput correlation analysis of Tomcat at workload 14,000 after upgrading the Tomcat JDK version. This figure shows that Tomcat no longer presents frequent transient bottlenecks compared to Figure 39(b). Specifically, the POIs in Figure 39(b) do not appear in Figure 41(a), which means the Tomcat JVM does not have long "freezing" periods after we upgrade the Tomcat JDK.

[6]JDK 1.6 uses garbage collection algorithms which support both parallel and concurrent garbage collection while JDK 1.5 by default uses a serial, stop-the-world collector.

Table 8: Partial P-states supported by the Xeon CPU of our machines

P-state	P0	P1	P4	P5	P8
CPU clock [MHz]	2261	2128	1729	1596	1197

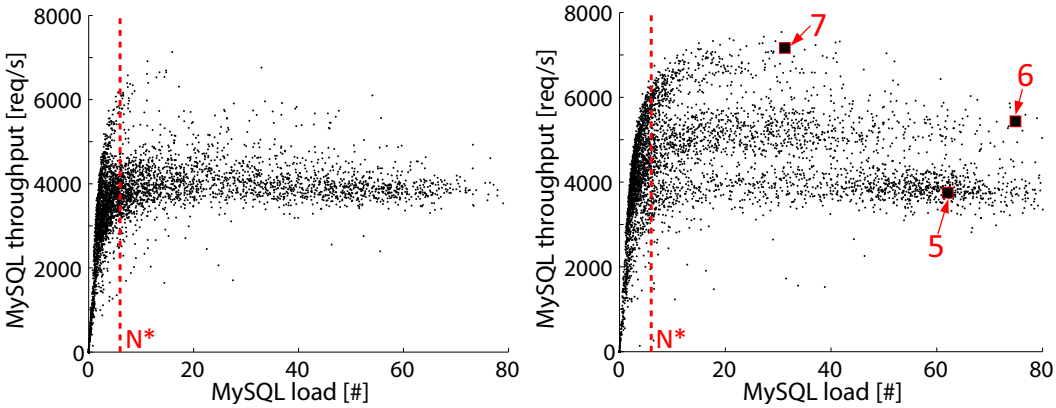
Figure 41(b) and 41(c) show the average system response time measured at every 50ms time intervals in the 3-minute experimental period before and after we upgrade Tomcat JDK version. These two figures show that the large response time fluctuations disappear after the JDK version upgrade, which shows that the system performance becomes more stable after we resolve the frequent transient bottlenecks in Tomcat.

4.4.3 Transient Bottlenecks Caused by Intel SpeedStep

The second case is the use of Intel SpeedStep technology which unintentionally causes transient bottlenecks, leading to the wide-range response time variations as we showed in Section 4.2.2. Intel SpeedStep allows the clock speed of a CPU to be dynamically adjusted (to different P-states) based on the real-time computing demands on a server in order to achieve a good balance between power usage and server performance; however, we found that the Dell's BIOS-level SpeedStep control algorithm cannot adjust the CPU clock speed quickly enough to match the real-time workload once the workload becomes bursty; the mismatch between CPU clock speed and real-time workload causes frequent transient bottlenecks that lead to the long-tail response time distribution as shown in Figure 31(c).

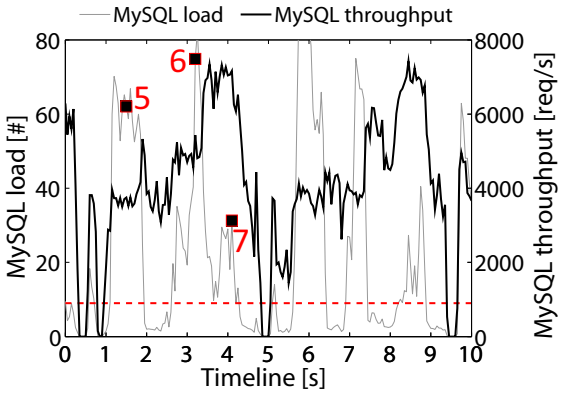
We enable the Intel SpeedStep support for MySQL in the BIOS settings to illustrate the mismatch problem. Table 8 shows a part of the P-states supported by our experimental machine CPU. This table shows that the CPU clock speed of the lowest P-state (P8) is nearly half of the highest P-state (P0). The experiments described here still keep the same 1L/2S/1L/2S configuration as in the previous sections with the only difference being the change in BIOS settings. We note that in all of the previous experiments, we disable the SpeedStep support in the BIOS settings of all our machines to simplify our analyses.

Figure 42 shows the fine-grained load/throughput analysis for MySQL at WL 8,000 and 10,000. As illustrated in Figure 31(c), the system already presents wide-range response time



(a) MySQL tier at WL 8,000

(b) MySQL tier at WL 10,000



(c) MySQL tier at WL 10,000

Figure 42: Fine-grained load/throughput(50ms) analysis for MySQL when CPU SpeedStep is enabled in MySQL. Figure 42(b) is derived from Figure 42(c), with 3-minute experimental data. Figure 42(a) shows one throughput trend when MySQL is temporarily bottlenecked, which indicates that MySQL chooses the lowest CPU clock speed when the workload is low. Figure 42(b) shows three throughput trends, which indicates that MySQL alternates among three CPU frequencies supported by Intel CPU SpeedStep as workload increases to 10,000.

variations at WL 8,000. Such variations are caused by the frequent transient bottlenecks in MySQL as shown in Figure 42(a). The interesting observation in Figure 42(a) is that though MySQL presents one main throughput trend (about 3700 req/s) when the load exceeds $N^{\star}$, there are many points above the main throughput trend, which contradicts our expectation of the shape of the main sequence curve. The comparison between Figure 42(a) and 42(b) reveals the cause. Since workload 8000 is relatively low, MySQL prefers to stay in P8-state in order to save power; however, MySQL is not responsive enough to scale-up to higher P-states to handle peak request rates from the upstream tiers in the system and thus

presents short-term congestions as shown in Figure 42(a). As workload increases to 10,000, Figure 42(b) shows that MySQL throughput presents three clear trends (about 3700 req/s, 5000 req/s, and 7000 req/s) when the corresponding load exceeds $N^\star$, which indicates that MySQL CPU alternates among three different P-states. For instance, the points labeled 5, 6, 7 show three time intervals when MySQL is temporarily congested but produces different throughputs. Point 5 indicates that MySQL stays in the lowest P8-state, point 6 indicates that MySQL stays in either P4- or P5-state, and point 7 indicates that MySQL stays in P0-state.

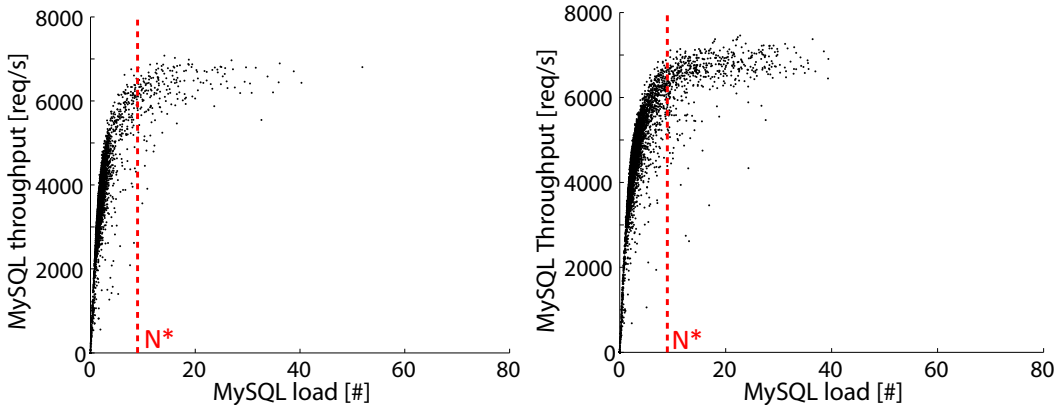
To illustrate when the mismatch of CPU clock speed and the real-time load on MySQL happens, Figure 42(c) shows the fine-grained MySQL load and throughput in a 10s experimental period at WL 10,000. The points labeled 5, 6, 7 correspond to the highlighted points in Figure 42(b), and show that in these three time intervals MySQL is temporarily congested but generates different throughputs. This figure illustrates the time lag of MySQL scaling-up to higher P-states, which causes frequent transient bottlenecks in MySQL.

4.4.4 Solution: Disable Intel SpeedStep in BIOS

Once detecting the frequent transient bottlenecks caused by the mismatch between CPU clock speed and bursty workload, we can resolve such bottlenecks by disabling the SpeedStep support in MySQL and let MySQL always stay in P0-state.

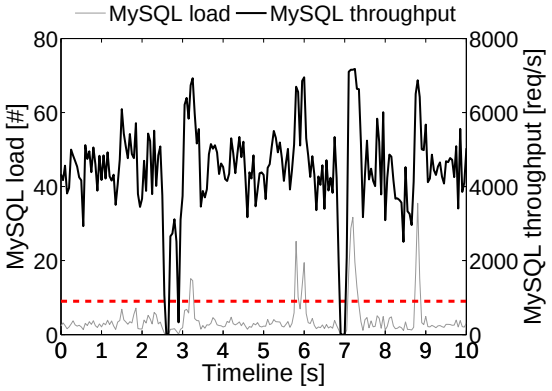
Figure 43 shows the fine-grained load/throughput analysis for MySQL at WL 8,000 and 10,000 after we disable the SpeedStep support in MySQL. Figure 43(a), 43(b) and 43(c) match back to Figure 42(a), 42(b) and 42(c), respectively. Since MySQL CPU always stays in P0-state, both Figure 43(a) and 43(b) show that MySQL only presents one throughput trend when load exceeds $N^\star$. More importantly, Figure 43(a) and 43(b) show that MySQL presents much less transient bottlenecks compared to the case shown in Figure 42(a) and 42(b) at WL 8,000 and 10,000. Figure 43(c) also shows that MySQL load is below $N^\star$ most of the time at WL 10,000, which suggests more stable performance of the system compared to Figure 42(c).

Further reduction of the transient bottlenecks in MySQL needs to either scale-out the



(a) MySQL at WL 8,000

(b) MySQL at WL 10,000



(c) MySQL at WL 10,000

Figure 43: Fine-grained load/throughput(50ms) analysis for MySQL when CPU SpeedStep is disabled in MySQL. Since MySQL always chooses to stay in the maximum CPU clock speed, the frequency of transient bottlenecks is significantly reduced by comparing Figure 43(a) and 43(b) with Figure 42(a) and 42(b).

MySQL tier (add more nodes to the MySQL tier) or scale-up MySQL (switch to a more powerful CPU).

4.5 Related Work

Techniques based on end-to-end request-flow tracing have been proposed in previous research for performance anomaly diagnosis. Magpie [19] and Pinpoint [27] focus on identifying anomalous requests that either have long response times or mutations of request-flow path by finding rare paths that differ greatly from others. Pip [67] identifies anomalous requests by comparing request-flows from actual behaviors and developer-expected behaviors. Spectroscope [70] proposes a similar monitoring infrastructure as Pip, but instead

of comparing request-flows between actual behaviors and developer-expected behaviors, it compares request-flows between "problem" periods and "non-problem" periods. Though detecting anomalous requests gives very useful hints to diagnose performance problem, they may fail to diagnose the root cause of anomalous requests in an n-tier system. A "anomalous" request may be slow not because of its own behavior, but because other requests were queued ahead of it [72, 78].

Analytical models have been proposed for bottleneck detection and performance prediction of n-tier systems. Urgaonkar [77] present a flexible queuing model for an n-tier application that determines how much resources to allocate to each tier of the application for the target system response time; however, this model is based on Mean Value Analysis (MVA), which has difficulties dealing with wide-range response time variations caused by bursty workloads and transient bottlenecks in the system. Mi et al. [54] propose a more sophisticated analytical model that predicts system performance based on bursty workloads. One challenge of this work is to precisely map the bursty characteristics of a workload to the queuing model with multiple service rates for each queue in the system. As shown in this paper, without fine-grained monitoring (sub-second level) granularity, the bursty characteristics of a workload and the potential transient bottlenecks as a result can be largely masked.

Software mis-configuration and failure detection of distributed system have been studied in [16,17,59]. Attariyan et al. [16,17] present a tool that locates the root cause of configuration errors by applying dynamic information flow analysis within a process (mainly) during runtime. Oliveira et al. [59] propose a mistake-aware management framework for protecting n-tier systems against operator mistakes by using the previous correct operations. All these works differ from our work in that they focus on faulty/anomalous behavior of system components rather than the performance problem.

4.6 Conclusion

We observed that the performance of an n-tier system may degrade significantly due to transient bottlenecks in component servers in the system. We proposed a novel bottleneck

detection method to detect these transient bottlenecks (Section 4.3), where the effectiveness of our approach is validated through the two case studies in Section 4.4. We found that transient bottlenecks can be caused by various factors at different levels of an n-tier application; for instance, JVM GC at the software level (Section 4.4.1) and Intel SpeedStep at the architecture level (Section 4.4.3). Solving these transient bottlenecks leads to significant performance improvements (Section 4.4.2 and 4.4.4). More generally, our work is an important contribution towards scaling complex n-tier applications under elastic workloads in cloud environments.

CHAPTER V

REMEDIES FOR LATENCY LONG-TAIL AND TRANSIENT BOTTLENECKS

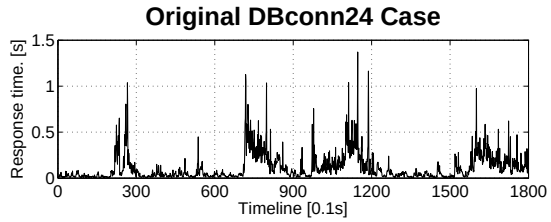
In this chapter, we present a systematic discussion of both specific and general remedies for reducing or avoiding the latency long-tail problem caused by transient bottlenecks. Although some causes for the very long response time (VLRT) requests can be "fixed" through some specific remedies (e.g., Java GC was streamlined from JVM 1.5 to 1.6), other VLRT requests arise from statistical coincidences such as VM consolidation (a kind of noisy neighbor problem) and cannot be easily "fixed". Using transient bottlenecks, we discuss the limitations of some potential solutions (e.g., making queues deeper through additional soft resource allocations causes bufferbloat) and describe generic remedies to reduce or bypass the queue amplification process (e.g., through the separation of short requests from resource-intensive requests to reduce queuing of short requests), regardless of the origin of transient bottlenecks.

5.1 Specific Solutions for Each Cause of VLRT Requests

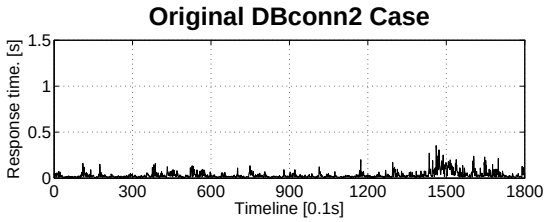
5.1.1 Solutions for VLRT Requests Caused by Java GC

When Java GC was identified as a source of VLRT requests (see Section 3.3.1), one of the first questions asked was whether we could apply a "bug fix" by changing the JVM 1.5 GC algorithm or implementation. Indeed this happened when JVM 1.6 replaced JVM 1.5. The new GC implementation was about an order of magnitude less demanding of CPU resources, and its impact became less noticeable at workloads studied in Section 4.4.2.

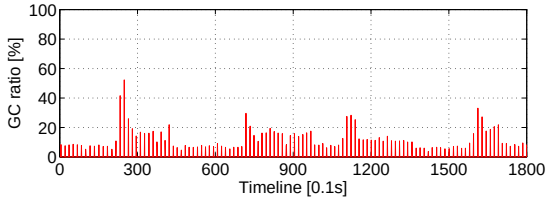
Another specific solution for Java GC is to restrict the number of processing threads in a Java-based server to avoid frequency Java GC caused by high concurrency in the server. The intuition behind this solution is that, given the same amount of workload, higher number of processing threads consumes more memory in the server and thus leaves larger memory footprint for garbage collection.



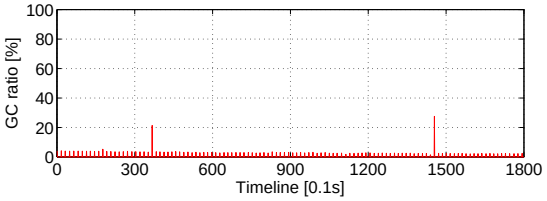
(a) Approximately instant response time in Apache web tier



(b) Approximately instant response time in Apache web tier



(c) JVM GC ratio on CJDBC, DBconn24 case



(d) JVM GC ratio on CJDBC, DBconn2 case

Figure 44: Response time stabilization by limiting the concurrency of the bottleneck tier in the system with 1L/2L/1S/2L configuration. The system keeps the same workload 5600 for the DBconn24 case (see (a) and (c)) and the DBconn2 case (see (b) and (d)).

The effectiveness of this solution is illustrated in Figure 44. The configuration of the system is 1L/2L/1S/2L (see Figure 1) where the CJDBC server CPU is the bottleneck of the system. We tune the number of processing threads in CJDBC by choosing the number of database connections in Tomcat. This is because each database connection in Tomcat corresponds to one processing thread in CJDBC; each time a Tomcat server establishes a connection to the CJDBC server, a thread is created by CJDBC to route the SQL queries received from Tomcat to one of the underlying database servers. We compare the Java GC activities and the end-to-end response time between two different number of database connections in Tomcat: DBconn24 and DBconn2, when the system is at the same amount of workload 5,400 clients. The CPU utilization can be find in Table 3.

Figure 44(a) and Figure 44(b) show the approximately instant end-to-end response time for the DBconn24 case and the DBconn2 case under workload 5500, respectively. These two figures show that the response time of the DBconn24 case exhibits much larger fluctuation than that of the DBconn2 case. Figure 44(c) and Figure 44(d) show the corresponding timeline graphs of JVM garbage collection ratio [1] in the CJDBC server for these two cases.

[1] JVM GC ratio means the ratio of JVM GC duration over each monitoring time window (e.g., 0.1 second).

Since the DBconn2 case performs less Java GC than the DBconn24 case does, the response time is more stable.

We note that limiting concurrency in the bottleneck tier is not always a good solution; too low concurrency in the bottleneck tier may under-utilize the hardware resource in the tier and degrade the overall system performance. A systematic way to choose a near-optimal concurrency for each tier in the system is discussed in our previous research [81].

5.1.2 Solutions for VLRT Requests Caused by Anti-Synchrony from DVFS

DVFS causing VLRT requests is due to anti-synchrony between workload bursts and DVFS power/speed adjustments (Section 3.3.2). We first use simulation to extend our experimental study and explore the impact of the default BIOS-level DVFS control (with a fixed-length DVFS adjustment period 500ms) on the controlled workload with different oscillation cycles. Then we show that the anti-synchrony could be avoided by changing (reducing) the control loop to adjust CPU clock rate more often, and thus disrupt the anti-synchrony for the default RUBBoS workload bursts.

5.1.2.1 Simulation Analysis of Anti-Synchrony between Workload and DVFS

Our study to quantify the impact (in terms of response time) for a given DVFS control escapes the classic assumptions of control systems. For example, instead of a fixed input workload model for which a control system can be designed with predictable maximum impact, the workload can vary arbitrarily. Consequently, we use the detailed simulator to find the maximum impact experimentally, which happens when the workload burst cycle and DVFS adjustment period are anti-synchronous (similar in length but out of phase).

The first step in our study is the implementation of an extended RUBBoS workload generator with fine-grain control over the period and intensity of bursty workloads [55]. The result is a bursty workload generator with two modes (high and low, e.g., 12,000 and 4,000 clients), plus controllable cycles between these two modes to simulate different burstiness levels in n-tier applications. A cycle consists of the generator running in one mode followed

JVM provides a logging function which records the starting/ending timestamp of every GC activity.

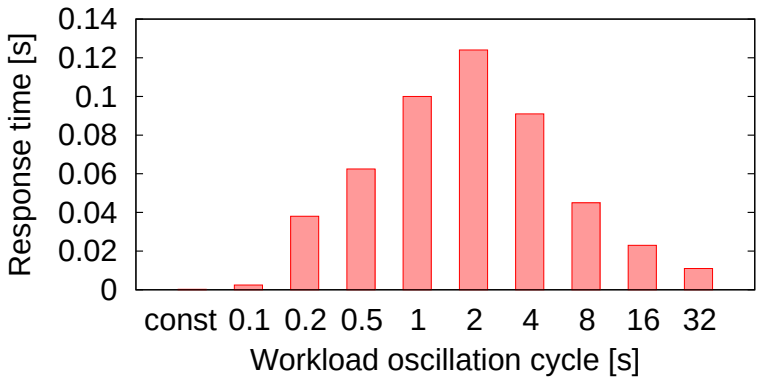
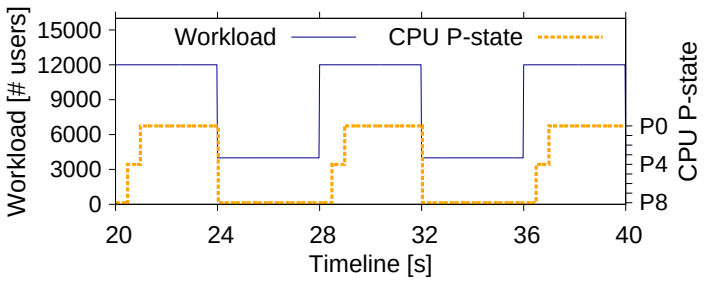


Figure 45: Simulation analysis of response time comparison among different workload oscillation cycles.

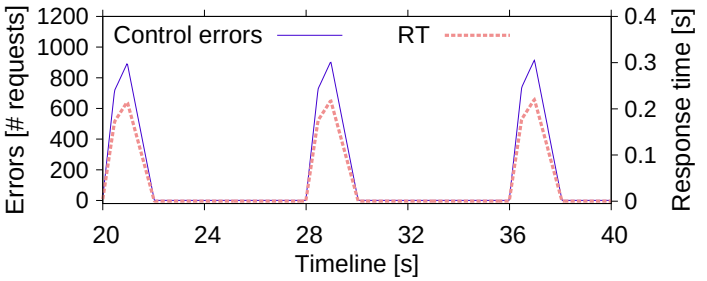
by a switch and running in the other mode. The generator then switches back to the original mode at the beginning of the next cycle. High burstiness is implemented as a short cycle and a steady workload can be implemented as an infinite cycle. In our experiments, we maintain the same average workload intensity level, e.g., 8,000 clients obtained by dividing the cycle evenly into half high (12,000 clients) and half low (4,000 clients). The cycle length is varied to generate different burstiness levels while maintaining the same workload intensity.

The high/low workload generator implementation enables a sensitivity study of response time as a function of workload burstiness. Figure 45 shows the average response time with DVFS-On with the same average workload intensity (8,000 clients) and high/low of 12,000/4,000 clients, but different workload burstiness cycles. The simulation data shows the system response time degraded the most from cycle time of 0.5sec to 4sec. From the control point of view, it is straightforward to explain the good response time for long cycles. The DVFS adjustments at 0.5sec intervals seem to be sufficient in handling the workload bursts that are longer than 4sec. Conversely, the very frequent cycles are also best handled by relatively slow control adjustments, since their behavior becomes increasingly similar to the average behavior at very high frequencies.

These conceptual explanations are confirmed by simulation data. Figure 46(a) shows the workload with oscillation cycle 8sec and the CPU clock rate (in P-state). When the workload intensity switches between high and low, the server takes two adjustment periods (about 1sec) to change between the lowest rate (P8) and the highest rate (P0). The adaptation



(a) Workload with oscillation cycle 8s and server CPU P-state. The adaptation delay of CPU clock rate happens when the workload switches from low to high.

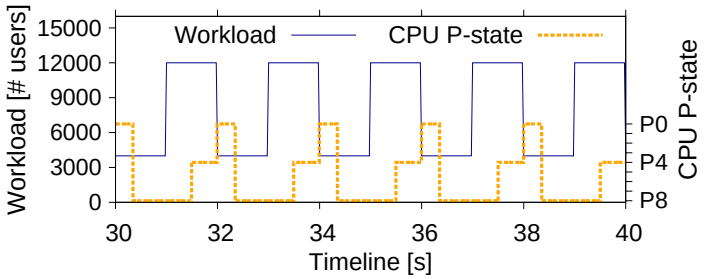


(b) Strong correlation between response time with control errors. The adaptation delay of CPU clock rate (see Figure 46(a)) causes large control errors, which in turn causes high peaks of the response time in the server.

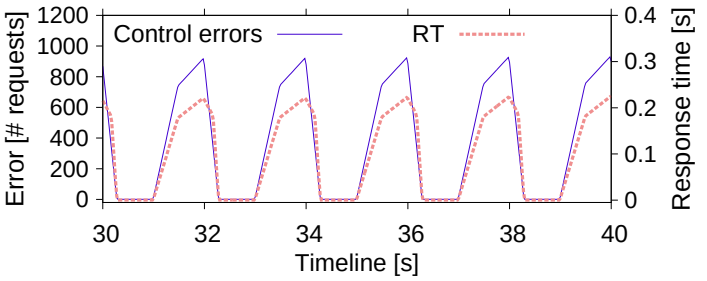
Figure 46: DVFS works well when the workload oscillation cycle is much longer than the DVFS adaptation period. Figure 46(a) and 46(b) show that though the control error caused by each adaptation delay of CPU clock rate is high, the frequency of adaptions is low.

time is relatively short and the system has a good match (high workload with high clock rate, and low workload with low clock rate) most of the time (7 out of 8sec). We define the queued requests in the server due to adaptation delay as *control errors*. The small error due to adaptation delay is shown in Figure 46(b). At the beginning of each cycle (e.g., the origin of the graph on the left), workload goes up to 12,000, causing a transient saturation of CPU and increase in response time. The temporary *Error* is shown in the graph and strongly correlated with the server response time increase. The saturation pushes DVFS to increase CPU clock rate, fixing the problem for the remainder 7/8 of the cycle.

On the other end of the spectrum, when the workload oscillation cycle is much smaller, the simulation confirms that the DVFS control period of 0.5sec works quite well, too. Figure 48(a) shows workload cycles and response time when the workload oscillates at 200ms cycles. The very fast workload oscillations actually reduce the *Error* since the



(a) Workload with oscillation cycle 2s and server CPU P-state. The adaptation delay of CPU clock rate covers the entire high workload period.



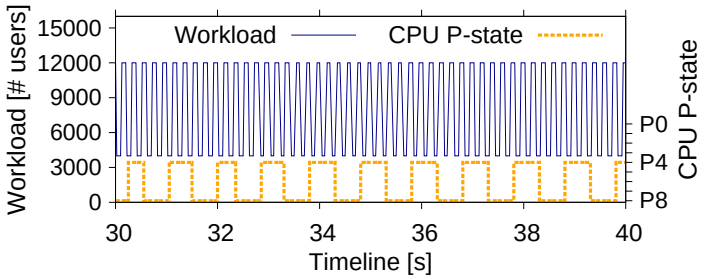
(b) Frequent high peaks of control errors and server response time due to the adaptation delay of CPU clock rate (see Figure 46(b))

Figure 47: DVFS causes the largest errors when the workload oscillation cycle is close to the DVFS adaptation period. Figure 47(a) shows that the server CPU always stays in the "wrong" P-state when the workload is high, thus the overall control errors reach the maximum as shown in Figure 46(b).

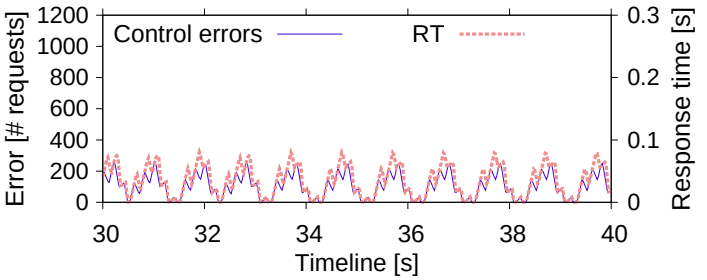
workload cycles back to a "correct" rate before the CPU reacts. Each DVFS adjustment (e.g., high clock rate) can match several periods of high workload rate before it switches to a lower clock rate. This is reflected in the error graph (Figure 48(b)).

For control system experts, it is perhaps unsurprising that the largest *Errors* appear when the workload oscillation cycles have lengths similar to the DVFS control cycle. This happens in the middle of spectrum. Figure 47(b) shows the response time and *Error* calculations for workload cycles at 2sec (the highest impact in Figure 45). The mismatch between workload burst cycles and DVFS adjustment periods is called anti-synchrony in analogy to anti-synchronous oscillatory systems. A concrete example shown in Figure 47(a) is at timeline 31 (and 33) when the workload cycle goes high just as the CPU clock rate has been slowed down.

We observe that by upper bound we do not mean the design of an adversarial workload



(a) Workload with oscillation cycle 200ms and server CPU P-state. The server CPU P-state is less sensitive to the rapid oscillation of workloads.



(b) No high peaks of control errors compared to Figure 46(b) and 47(b), which leads to more stable response time.

Figure 48: DVFS works well when the workload oscillation cycle is much shorter than the DVFS adaptation period. Figure 48(a) shows that the DVFS controller is more robust to the noise of a rapidly changing workload due to the relatively large adjustment period(500ms), which leads to much smaller accumulated errors and thus more stable response time as shown in Figure 48(b).

that would achieve the theoretical maximum *Error* (i.e., exact anti-synchronous cycles against the DVFS adjustment periods), which is an interesting exercise but beyond the scope of this paper. In this study, we aim to find the upper bound for bursty workloads that are reasonably regular and likely to happen in real world applications. We believe that alternative bursty workload models would yield substantially similar experimental results due to the necessary matching with the same DVFS adjustment periods.

5.1.2.2 Workload-Sensitive Adaptive Control

Since workload burst cycles may vary, any fixed-length DVFS adjustment period will remain vulnerable to anti-synchrony. Consequently, a workload-sensitive adjustment method seems a better solution. Direct observations of workload intensity is challenging in n-tier systems, since there are significant and mutually-dependent variations at each tier. Instead, we

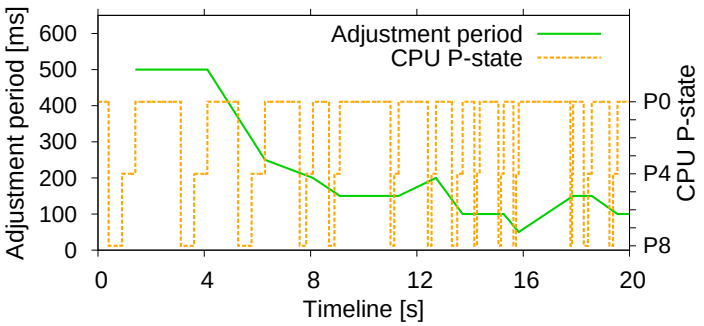


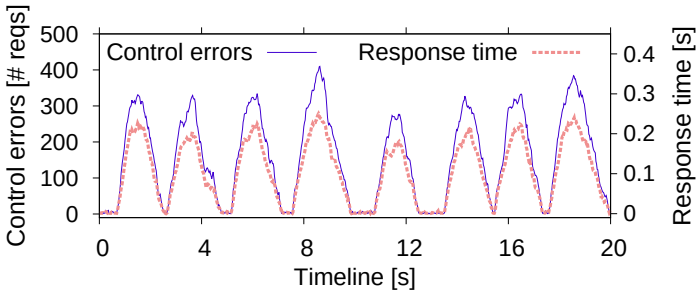
Figure 49: Adjustment period variation in the adaptive controller when server is at workload 11,000.

choose to add a second level adaptive control on the DVFS itself.

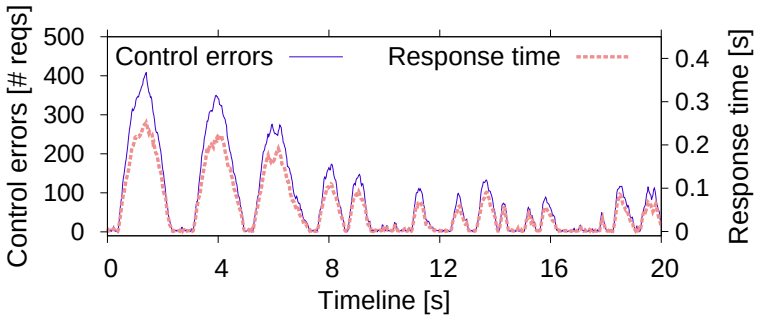
Our design considers a classic (fixed-period) DVFS as the system to be controlled. The main observable of interest is the CPU P-state switch. We consider the change from P8 (slowest clock rate) to P0 (fastest clock rate) as an indication of confirmed workload burst. More generally, the interval between consecutive such switches can be considered a reasonable estimate of the workload burst cycle. For example, Figure 46(a) and Figure 47(a) shows that the CPU switches from P8 to P0 (P4 in between) frequently and the period of workload burst is every 8 seconds and 2 seconds, respectively. The second level controller uses the observed workload burst cycle to predict the onset of next workload burst. Although burst cycles may change over time, we assume that burst cycles can be estimated by a linear function in the neighborhood of an operating point. We adopt a simple moving-average (MA) model [83] to predict the workload burst cycle as shown in the Equestion 3:

$$B_{i+1} = \frac{1}{k} \sum_{j=i-k+1}^{i} B_j \tag{3}$$

The MA model assumes the short-term dependencies between successive burst cycles. B_{i+1} denotes the length of the next workload burst cycle. k refers to the previous k burst cycles remembered: the larger the k, the more past P-state switches will be taken into account. We estimated the model offline using least-squares based on methods in the Matlab System ID Toolbox to fit the input-output data collected from simulation. The model is evaluated using the r^2 metrics defined in Matlab as a goodness-of-fit measure. In general, the r^2 value indicates the percentage of variation in the output captured by the



(a) Server response time and control errors of the original controller



(b) Server response time and control errors of the adaptive controller

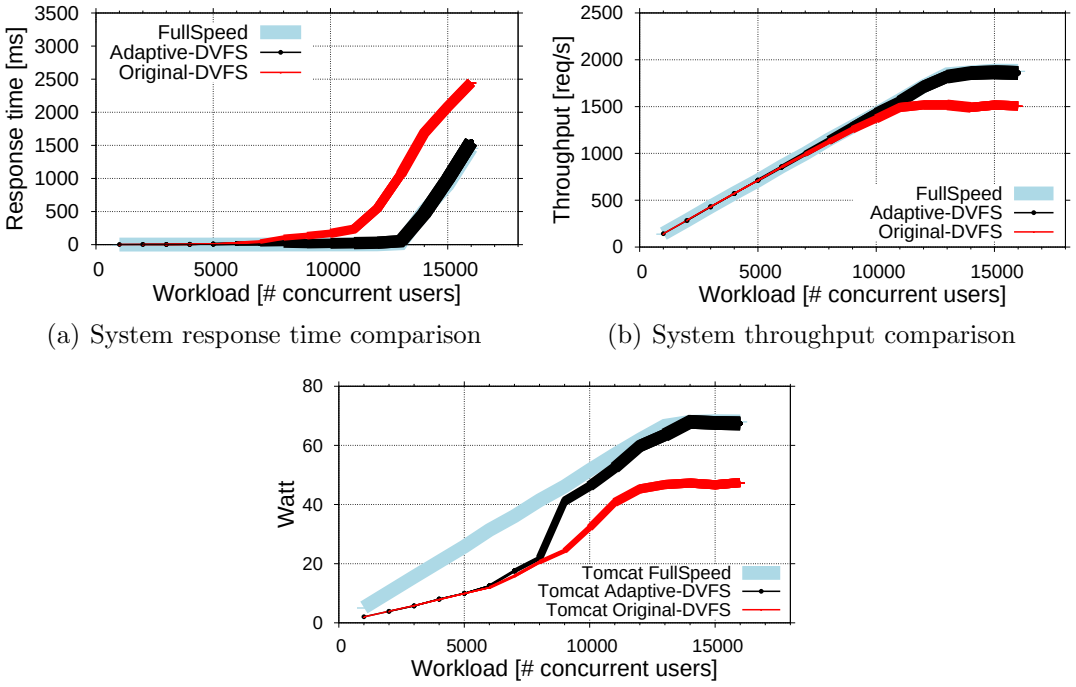
Figure 50: Comparison between the adaptive DVFS controller and the original DVFS controller. Figure 50(b) shows that the latter case is more effective to reduce the control errors and stablize the server response time.

model. In our case, the r^2 is 0.912, which indicates a good fit of the model.

Once we obtain the predicted workload burst cycle, we adjust the length of adjustment period accordingly to reduce the DVFS negative impact. For example, we can set the length of adjustment period to be proportionally smaller than the workload burst cycle as shown in the following equation:

$$T_i = \begin{cases} T_{lb} & \text{if } (B_i/n) < T_{lb} \\ T_{ub} & \text{if } (B_i/n) > T_{ub} \\ B_i/n & \text{otherwise} \end{cases} \quad (4)$$

Both T_{lb} and T_{ub} are thresholds that prevent the new adjustment period to be either too small (significant P-state switching overhead) or too large (significant performance degradation caused by prediction error). n sets the distance between the workload burst cycle and the new adaptation period (which is 2 adjustment periods in our case). As shown



(a) System response time comparison

(b) System throughput comparison

(c) Tomcat power consumption comparison

Figure 51: Comparison among three different DVFS policies in the context of n-tier applications. The adaptive DVFS controller achieves better balance between performance and power usage than the other two.

in Figure 45, the larger distance is set between these two metrics, the better performance of the system. A setting of 4-8 times larger would typically leads to fairly good results. In our evaluation, we set T_{lb}, T_{ub}, and n to be 50ms, 500ms, and 10. Thus, the burst cycle is always 5 times larger than the adaptation period in our adaptive controller.

Figure 50 shows the effectiveness of the adaptive DVFS controller compared to the original controller with fixed adjustment periods on a server at workload 11,000. Figure 49 shows that the length of the adaptation period varies over time due to the changes of the workload bursty cycle in the adaptive controller case. Figure 50(b) shows the *Error* caused by anti-synchrony between DVFS adjustment period and workload burst cycle. The server response time (and *Error*) become smaller and more stable over time due to the workload-sensitive adaptive changes of the DVFS adjustment period. Figure 50(a) shows high *Error* and response time of the original DVFS controller compared to our two-level adaptive DVFS controller.

Figure 51 further shows the effectiveness of the adaptive DVFS controller in n-tier applications. We apply the adaptive DVFS controller to the 4-tier system (see Figure 11) in simulation. Figure 51(a) and 51(b) show that the adaptive DVFS controller achieves the similar response time and throughput to the FullSpeed case on the entire workload range, which suggests that the adaptive DVFS controller doesn't cause significant performance loss. Since in simulation we are unable to directly measure the power consumption of each server, we adopt a simple model [43] to estimate the CPU power consumption as a function of CPU P-state distribution and CPU utilization:

$$P = C * R_{active} * V^2 * f + P_{static} \tag{5}$$

Here, C is a constant related to the capacitance of transistor gates, R_{active} is the CPU utilization, V is the operating voltage, and f is the CPU frequency. From the specification of the Xeon CPU model in our experiments we know the CPU frequency and voltage at each P-state [2]. To simplify the analysis, we denote C to be 1 and P_{static} to be 0 since we only consider the CPU dynamic power consumption. Thus given the measured CPU utilization and P-state of each server in simulation, we are able to estimate the power consumption of each server at each workload.

Figure 51(c) shows the adaptive DVFS controller is able to save the similar amount of power as the original DVFS controller does before workload 8,000 while it gradually merges to the FullSpeed case as workload continue to increase. The width of each of the three lines is proportional to the average CPU clock rate. Overall Figure 51 shows that the adaptive DVFS controller achieves better balance between performance and power consumption than the other two policies. Evaluating our solution in real implementation would naturally be the future work of this paper.

As new sources of VLRT requests such as VM consolidation (Section 3.3.3) continue to be discovered, and suggested by previous work [32, 58], the "bug fix" approach may be useful for solving specific problems, but it probably would not scale, since it is a temporary

[2] E.g., 1.197 GHz/0.750V at P8-state, 2.26GHz/1.350V at P0-state

remedy for each particular set of configurations with their matching set of workloads. As workloads and system components (both hardware and software) evolve, VLRT requests may arise again under a different set of configuration settings. It will be better to find a more general approach to resolve entire classes of problems that cause VLRT requests.

5.2 Solutions for Transient Bottlenecks

We will discuss potential and general remedies using transient bottlenecks as a simple model, regardless of what caused the VLRT requests (three very different causes of transient bottlenecks were described in Section 3.3). For this discussion, a transient bottleneck is a very short period of time (from tens to hundreds of milliseconds) during which the CPU remains busy and thus continuously unavailable for lower priority threads and processes at kernel, system, and user levels. The usefulness of the transient bottleneck model in the identification of causes of VLRT requests has been demonstrated in Section 3.3, where VLRT requests were associated with transient bottlenecks in three different system layers.

As we consider the design and evaluation of general remedies for transient bottlenecks, the knowledge of the actual sources that caused the transient bottleneck is useful and relevant, but not necessary. This observation is consistent with Dean's paper on building latency tail-tolerant systems [32]. Similarly, if a bottleneck becomes persistent (longer than seconds), then it becomes a more traditional problem where traditional techniques (admission control and load balancing) would apply. Therefore, we will focus on the middle three steps of the micro-event analysis done in Section 3.3: retransmitted requests, queue amplification, and transient bottlenecks.

In contrast to the effect-to-cause analysis in Section 3.3, the following discussion of general remedies will follow the chronological order of events, where transient bottlenecks happen first, causing queue amplification, and finally retransmitted VLRT requests. For concreteness, we will use the RUBBoS n-tier application scenario; the discussion applies equally well to other mutually-dependent distributed systems.

First, we will consider the disruption of transient bottleneck formation. From the description in Section 3.3, there are several very different sources of transient bottlenecks,

including system software daemon processes (e.g., Java GC), predictable control system interferences (e.g., DVFS), and unpredictable statistical interferences (e.g., VM co-location). A general solution that is independent of any causes would have to wait for a transient bottleneck to start, detect it, and then take remedial action to disrupt it. Given the short lifespan of a transient bottleneck, its reliable detection becomes a significant challenge. Using a control system terminology, if we trigger the detection too soon (e.g., after 1 millisecond of saturation) we have fast but unstable response. Similarly, if we wait too long in the control loop (e.g., tens of milliseconds), we may have more stable response but the damage caused by transient bottleneck may have already been done. This argument does not prove that the cause-agnostic detection and disruption of a transient bottleneck is impossible, but it is a serious research challenge.

Second, we will consider the disruption of the queue amplification process. A frequently asked question is whether lengthening the queues in servers (e.g., increasing TCP buffer size or thread pool size in Apache and Tomcat) can disrupt the queue amplification process. There are several reasons for large distributed systems to limit the depth of queues in components. At the network level (e.g., TCP), large network buffer size causes problems such as bufferbloat [41], leading to long latency and poor system performance. At the software systems level, over allocation of threads in web servers can cause significant overhead [81,82], consuming critical bottleneck resources such as CPU and memory and degrade system performance. Therefore, the queue lengths in servers should remain limited.

On the other hand, the necessity for limitation in server queues does not mean that queue amplification is inevitable. An implicit assumption in queue amplification is the synchronous request/response communication style in current n-tier system implementations (e.g., with Apache and Tomcat). It is possible that asynchronous servers (e.g., nginx [6]) may behave differently, since it does not use threads to wait for responses and therefore it may not propagate the queuing effect further upstream. This interesting area (changing the architecture of n-tier systems to reduce mutual dependencies) is the subject of ongoing active research.

Another set of alternative techniques have been suggested [32] to reduce or bypass

queue-related blocking. An example is the creation of multiple classes of requests [78], with a differentiated service scheduler to speed up the processing of short requests so they do not have to wait for VLRT requests (see Section 5.2.1 for more details). A related idea is to break heavier tasks into a sequence of small sub-tasks, which may benefit more from a multi-class scheduler. Some applications allow semantics-dependent approaches to reduce the latency long tail problem. For example, (read-only) web search queries can be sent to redundant servers so VLRT requests would not affect all of the replicated queries. These alternative techniques are also an area of active research.

Third, we will consider the disruption of retransmitted requests due to full queues in servers. Of course, once a packet has been lost it is necessary to recover the information through retransmission. Therefore, the question is about preventing packet loss. The various approaches to disrupt queue amplification, if successful, can also prevent packet loss and retransmission. Therefore, we consider the discussion on disruption of queue amplification to subsume the packet loss prevention problem. A related and positive development is the change of the default TCP timeout period from 3 seconds to 1 second in the Linux kernel [45].

Fourth, we return to the Gartner report on average data center utilization of 18% [74]. An empirically observed condition for the rise of transient bottlenecks is a moderate or higher average CPU utilization. In our experiments, transient bottlenecks start to happen at around 40% average CPU utilization. Therefore, we consider the reported low average utilization as a practical (and expensive) method to avoid the transient bottleneck problem. Although more research is needed to confirm this conjecture, low CPU utilization levels probably help prevent transient bottleneck formation as well as queue formation and amplification.

5.2.1 Transaction Level Priority-Based Scheduling

Many previous research efforts [44, 49, 71] show that the performance of a single web server can be dramatically improved via a kernel-level modification by changing the scheduling policy from the standard FAIR (processor-sharing) scheduling to SJF (shortest-job-first)

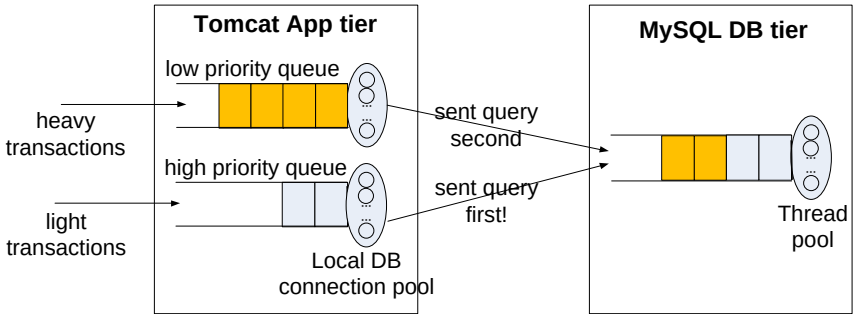


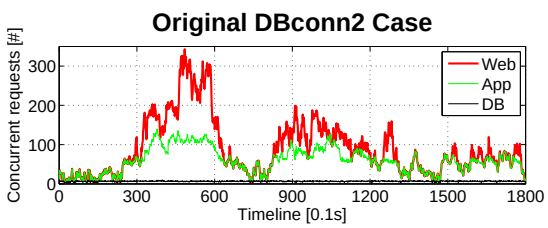
Figure 52: Illustration of applying CTP scheduling policy across tiers (only 2 servlets shown).

scheduling. However, for more complex n-tier systems where a completion of a client transaction involves multiple interactions among tiers, the best OS level scheduling policy may increase the overall transaction response time. This is because the operating system of each individual server in an n-tier system cannot distinguish heavy transactions from light transactions without application level knowledge. A transaction is heavier than the other one may just because it involves more interactions among tiers than those of the other one, while the processing time of each interaction may be even shorter than the counterpart (See Figure 10). Thus, applying SJF scheduling policy to the operating system of each tier may actually delay the processing of the application-level light transactions. Given such observation, we propose the following transaction level priority-based scheduling:

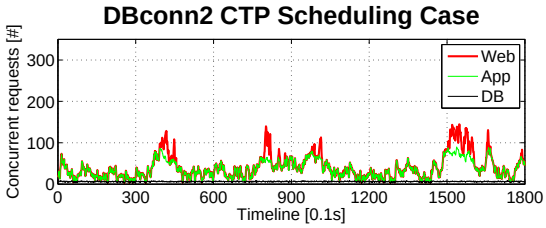
We need to give higher priority to light transactions than heavy transactions to minimize the total amount of waiting time in the whole n-tier system. We need to schedule transactions in an upper tier which can distinguish light transactions from heavy transactions.

This solution is essentially an extension of applying the SJF scheduling policy in the context of n-tier systems. Suppose the MySQL tier is the bottleneck tier; applying SJF scheduling policy to MySQL through the kernel-level modification may not reduce the overall system response time because MySQL cannot distinguish application level heavy transactions and light transactions. Thus we need to schedule transactions in an upper tier that can make such distinction in order to apply SJF scheduling policy properly in an n-tier system. We define such scheduling as cross-tier-priority (CTP) based scheduling.

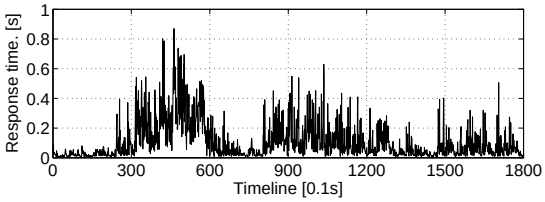
Figure 52 illustrates how to apply the CTP scheduling to a simple two-tier system. This



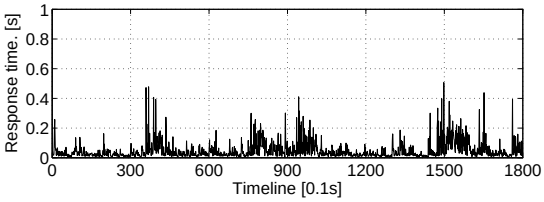
(a) Approximately instant # of concurrent requests in each tier



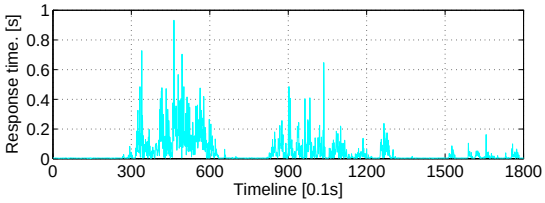
(b) Approximately instant # of concurrent requests in each tier



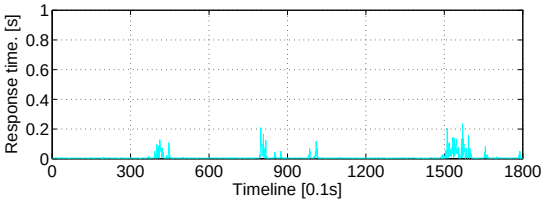
(c) Approximately instant response time in Apache web tier



(d) Approximately instant response time in Apache web tier



(e) Approximately instant response Time for StoryOfTheDay requests in Apache web tier



(f) Approximately instant response Time for StoryOfTheDay requests in Apache web tier

Figure 53: Response time stabilization by applying CTP scheduling in 1L/2L/1L configuration in WL 5800.

figure shows only requests for two servlets (the RUBBoS browse-only workload consists of requests for eight servlets): ViewStory (heavy) and StoryOfTheDay (light). Once ViewStory requests and StoryOfTheDay requests reach the Tomcat App tier at the same time, we give StoryOfTheDay requests higher priority to send queries to MySQL. In this case the waiting time of the light StoryOfTheDay transactions can be reduced and the overall waiting time for all transactions is reduced [3].

Figure 53 shows the response time stabilization by applying the CTP scheduling to a three-tier system (1L/2L/1L with DBconn2) in workload 5800. Under this configuration, the MySQL CPU is the bottleneck in the system. Figure 53(a) and 53(c) show the results of the

[3]Heavy transactions are only negligibly penalized or not penalized at all as a result of SJF-based scheduling [44].

original RUBBoS implementation (using the default OS level scheduling) and Figure 53(b) and 53(d) show the results after the CTP scheduling is applied to the Tomcat App tier and the MySQL DB tier (see Figure 52).

Figure 53(a) and Figure 53(b) show the number of concurrent requests in each tier of the three-tier system for these two cases. Although in both cases the number of concurrent requests in the MySQL tier is very small (around eight), the fluctuations of the number of concurrent requests in the Tomcat App tier and the Apache web tier are much higher in the original case than those in the CTP scheduling case. This is because in the original case more light requests are queued in the upper tiers due to the increased waiting time of light requests in the MySQL tier.

Figure 53(c) and Figure 53(d) show that the approximately instant response time in the Apache web tier in the original case has much larger fluctuations than that in the CTP scheduling case, which validates that CTP scheduling actually reduces the overall waiting time of all transactions in the system. In fact the high peaks of response time in these two figures perfectly matches the high peaks of the number of queued requests in upper tiers as shown in Figure 53(a) and Figure 53(b).

CHAPTER VI

RELATED WORK

Latency has received increasing attention in the evaluation of quality of service provided by computing clouds and data centers [15, 64, 69, 78, 80]. Specifically, the long-tail latency is of particular concern for mission-critical web-facing applications [13, 14, 32, 49, 84]. On the solution side, Dean et al. [32] described their efforts to mitigate tail latency in Google's interactive applications. These bypass techniques are effective in specific applications or domains, contributing to an increasingly acute need to improve our understanding of the general causes for the VLRT requests.

The potential causes for the performance problem of web applications have been studied extensively in many previous research. Dean et al. [32] outlined several potential causes for the tail latency problem of Google's large scale interactive applications. Examples include, but not limit to shared resources (such as CPU cores, processor caches) by different applications running on top of the same set of machines, background daemons, maintenance activities, and several hardware trends such as power limits for modern CPUs, garbage collection for solid-state storage devices, and power-saving modes in many types of devices. Software mis-configuration for the failure of distributed systems have been studied in [16,17, 59]. Workloads characteristics such as burstiness or request type mix-ratio for the impact of system performance have been studied in [24, 30, 40, 46, 53, 55, 56, 73]. In addition, soft resource (e.g., threads, database connections) allocation has been discussed as an important source for unpredictable performance in [10, 21, 35, 39, 51, 60, 61, 63, 66, 82, 85].

Aggregated statistical analyses over fine-grained monitored data have been used to infer the appearance and causes of long-tail latency [31, 49, 80]. Li et al. [49] measure and compare the changes of latency distributions to study hardware, OS, and concurrency-model induced causes of tail latency in typical web servers executing on multi-core machines. Wang

et al. [80] propose a fine-grained correlation analysis between a server's throughput and concurrent jobs in the server to infer the server's real-time performance state. Cohen [31] use a class of probabilistic models to correlate system-level metrics and threshold values with high-level performance states. Our work leverages the fine-grain data, but we go further in using micro-level timeline event analysis to link the various causes to VLRT requests.

Our work makes heavy use of data from fine-grained monitoring and profiling tools [4,7]. Related techniques have been proposed to help detect a performance problem and identify symptoms associated with the problem [23,25,49,65,68]. For example, Collectl [4] provides the ability to monitor a broad set of system level metrics such as CPU and I/O operations at millisecond-level granularity. Chopstix [23] continuously collects profiles of low-level OS events (e.g., scheduling, L2 cache misses, page allocation, locking) at the granularity of executables, procedures and instruction. Li et al. [49] propose a fine-grained timestamping technique to measure how much time a request spends in different parts of the server OS. We use these tools when applicable.

Techniques based on end-to-end request-flow tracing have been proposed for performance anomaly diagnosis [12,16,27,36,67,70], but usually for more stable and longer phenomena. X-ray [16] instruments binaries as applications execute and uses dynamic information flow tracking to estimate the likelihood that a block was executed due to each potential root cause for the performance anomaly. Fay [36] provides dynamic tracing through use of runtime instrumentation and distributed aggregation within machines and across clusters for windows platform. Aguilera et al. [12] infer causal paths between component servers in a distributed system and attribute delays to specific nodes. Pip [67] detects anomalous requests by comparing request-flows from actual behaviors with developer-expected behaviors. Spectroscope [70] is similar to Pip, but Spectroscope compares request-flows between "problem" periods and "non-problem" periods for identifying anomalous requests.

Analytical models have been proposed for performance analysis and prediction of n-tier systems. Magpie [19] extracts the component control flow and resource consumption of each request to build a workload model for performance prediction. Urgaonkar [77] present a flexible queuing model for an n-tier application that determines how much resources to

allocate to each tier of the application for the target system response time; Cohen [31] use a class of probabilistic models to correlate system-level metrics and threshold values with high-level performance states. Though they have been shown to be accurate when the system resource utilization is low, they may fail when the system has latency long-tail problem caused by frequent transient bottlenecks.

CHAPTER VII

CONCLUSION AND FUTURE WORK

Our research is motivated by the essential requirement of simultaneously achieving good performance and high utilization for cost efficiency in cloud computing environments. High utilization through virtualization and hardware resource sharing is critical for both cloud providers and cloud consumers to reduce management and infrastructure costs (e.g., energy cost, hardware cost) and increases cost-efficiency. For example, doubling the utilization of a typical datacenter (currently 18% on average) may potentially double the total revenue of the datacenter. Unfortunately, achieving good performance for web-facing applications at high resource utilization remains an elusive goal. Both practitioners and researchers have experienced the latency long-tail problem in clouds during periods of high utilization. My research aims to achieve good performance of large scale web-facing applications (e.g., Google, Facebook, Amazon, etc.) running at high utilization.

Our research shows that transient bottlenecks are an important contributing factor to the latency long-tail problem. Transient bottlenecks are bottlenecks with a short lifespan on the order of tens of milliseconds. Though short-lived, transient bottleneck can cause a long-tail response time distribution that spans a spectrum of 2 to 3 orders of magnitude, from tens of milliseconds to tens of seconds, due to the queuing effect propagation and amplification caused by complex inter-tier resource dependencies in the system (see Chapter 2).

Transient bottlenecks can arise from a wide range of factors at different system layers. For example, we have identified transient bottlenecks caused by CPU DVFS control at the architecture layer, Java garbage collection at the system software layer, and interferences among virtual machines (VM) in VM consolidation at the VM layer. Applying a micro-level event analysis on extensive experimental data collected from fine-grain monitoring of n-tier application benchmarks, we demonstrate that the latency long tail problem is explicitly linked to the three identified causes for transient bottlenecks. Specifically, the

micro-level event analysis shows the VLRT requests are coincidental to transient bottlenecks in various servers, which in turn amplify queuing in upstream servers, quickly leading to TCP buffer overflow and request retransmission, causing VLRT requests of several seconds (see Chapter 3).

We propose a novel bottleneck detection method transient bottleneck detection method in Chapter 4, which is sensitive enough to detect transient bottlenecks at millisecond level with negligible monitoring overhead. Our method uses a passive network tracing facility that timestamps all network packets going through an n-tier system at microsecond granularity. By combining the fine-grained monitoring data and a sophisticated analytical method to analyze monitoring data, we are able to detect and study transient bottlenecks with duration as short as 50ms. There are two advantages of our method. The first one is that it is completely independent of specific resource saturation measurements, thus transient bottlenecks with varied causes even in different system layers can be identified. The second one is that the monitoring is mainly at the network switch which supports the port mirroring function. Thus the performance overhead for the target runtime application is negligible (assume the network switch is not the bottleneck).

We discuss several approaches to remedy the emergence of VLRT requests caused by transient bottlenecks, including cause-specific "bug-fixes" and more general solutions to reduce queuing based on the transient bottleneck model that will work regardless of the origin of VLRT requests (Chapter 5). We believe that our study of transient bottlenecks uncovered only the "tip of iceberg". There are probably many other important causes of transient bottlenecks such as background daemon processes that cause "multi-millisecond hiccups" [32]. Our discussion in Section 3.4 suggests that the challenge to find effective remedies for transient bottlenecks has only just begun.

7.1 Future Work

7.1.1 Extension of Dissertation Work

In my dissertation I have introduced the transient bottlenecks caused by different factors from different system layers. As we can see that the essential problem of a transient bottleneck is the temporarily resource shortage (e.g., temporarily CPU saturation) during the bottleneck period, which causes requests to queue in the bottlenecked server and potentially in the upper tiers due to inter-tier resource dependencies. Thus a transient bottleneck can be modeled as a temporary resource shortage that causes requests to queue in the system, regardless of the root cause of the transient bottleneck. Then a natural question comes after this observation: what exactly is the relationship between the duration of a transient bottleneck and its negative impact on the end-to-end response time variations? This is an important question since transient bottleneck is usually at the time scale of tens to few hundreds of milliseconds, which is apparently too short for application level measurements (often done at periods of multiple seconds) not to be taken into consideration when measuring application level performance. Furthermore, even if two transient bottlenecks have the same length, do they have the same negative impact on the end-to-end response time if they occur in the servers from different tiers? If not, which tier is the most influential tier for the entire system performance? I would expect the answer is non-trivial and highly dependent on the workload characteristics (e.g., CPU or I/O intensive, or interaction ratio among tiers). Overall, understanding and modeling the impact of transient bottlenecks on the end-to-end response time can significantly improve our web application performance management in the cloud.

Also the discussion in Section 3.4 suggests that the challenge to find effective remedies for transient bottlenecks has only just begun. I would like to explore more cause-specific remedies and general solutions, regardless of the origin of transient bottlenecks, to reduce or avoid the latency long-tail problem.

7.1.2 Looking Beyond: Autonomic Cloud Application Management

My final research goal is the autonomic management of applications in cloud environments, encompassing application deployment, monitoring, evaluation, and evolving with the goal of high performance and high utilization. Cloud applications may evolve frequently during its lifetime due to changes such as increases in workload, feature enrichment, bug fixing, and implementation of new functionalities. Current approaches to cloud application deployment, monitoring, evaluation and reconfiguration are mostly done manually and the process is time-consuming due to the complexity of each of these four phases. For example, monitoring not only includes application level metrics (e.g., response time, throughput) and system level metrics (e.g., CPU or I/O utilization), but also various event logs generated by each component node of distributed systems. Some monitoring data, especially event logs, are difficult to read and to analyze automatically due to their free form nature and lack of inter-node references. I intend to develop a general usage model of the heterogeneous monitoring data to facilitate the storing and analyzing the data. In addition, reconfiguration is another challenge for autonomic system management since there may be thousands of tuning knobs in a cloud application. In fact, misconfiguration is considered as one dominating cause for application failures or poor performance. My previous research shows that many tuning knobs in a distributed system are inter-dependent. I expect that it is possible to develop intelligent techniques that extract the dependencies among various tuning knobs and use it to facilitate the automatic reconfigurations. There are many other challenging problems towards autonomic cloud application management, such as performance prediction for system reconfiguration and automatic bottleneck detection. I plan to conduct systematic studies in these new areas.

REFERENCES

[1] *Dstat: Versatile Resource Statistics Tool.* "http://dag.wieers.com/home-made/dstat/", 2010.

[2] *Fujitsu SysViz: System Visualization.* "http://www.google.com/patents?id=0pGRAAAAEBAJ&zoom=4&pg=PA1#v=onepage&q&f=false", 2010.

[3] "The AJP connector." "http://tomcat.apache.org/tomcat-7.0-doc/config/ajp.html".

[4] "Collectl." "http://collectl.sourceforge.net/".

[5] "Java SE 6 performance white paper." "http://java.sun.com/performance/reference/whitepapers/6_performance.html".

[6] "NGINX." "http://nginx.org/".

[7] "Oprofile." "http://oprofile.sourceforge.net/".

[8] "RUBBoS: Bulletin board benchmark." "http://jmob.ow2.org/rubbos.html".

[9] Abdelzaher, T., Diao, Y., Hellerstein, J., Lu, C., and Zhu, X., "Introduction to control theory and its application to computing systems," SIGMETRICS Tutorial, 2008.

[10] Aberer, K., Risse, T., and Wombacher, A., "Configuration of distributed message converter systems using performance modeling," in *Proceedings of 20th International Performance, Computation and Communication Conference (IPCCC'2001), IEEE Computer Society*, Citeseer, 2001.

[11] Adler, S., "The slashdot effect: an analysis of three internet publications," *Linux Gazette*, vol. 38, p. 2, 1999.

[12] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A., "Performance debugging for distributed systems of black boxes," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pp. 74–89, 2003.

[13] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M., "Data center TCP (DCTCP)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, pp. 63–74, 2010.

[14] Alizadeh, M., Kabbani, A., Edsall, T., Prabhakar, B., Vahdat, A., and Yasuda, M., "Less is more: Trading a little bandwidth for ultra-low latency in the data center," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, pp. 253–266, 2012.

[15] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and others, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[16] Attariyan, M., Chow, M., and Flinn, J., "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, pp. 307–320, 2012.

[17] Attariyan, M. and Flinn, J., "Automating configuration troubleshooting with dynamic information flow analysis," in *OSDI '10*, (Vancouver, BC, Canada), 2010.

[18] Bahl, P., Chandra, R., Greenberg, A., Kandula, S., Maltz, D. A., and Zhang, M., "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *SIGCOMM'07*.

[19] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R., "Using magpie for request extraction and workload modelling," in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, pp. 259–272, 2004.

[20] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pp. 164–177, 2003.

[21] Beltran, V., Torres, J., and Ayguade, E., "Understanding tuning complexity in multithreaded and hybrid web servers," in *IPDPS'08: IEEE International Parallel and Distributed Processing Symposium*, 2008.

[22] Bertoli, M., Casale, G., and Serazzi, G., "JMT: performance engineering tools for system modeling," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 10–15, 2009.

[23] Bhatia, S., Kumar, A., Fiuczynski, M. E., and Peterson, L., "Lightweight, high-resolution monitoring for troubleshooting production systems," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pp. 103–116, 2008.

[24] Bodik, P., Fox, A., Franklin, M. J., Jordan, M. I., and Patterson, D. A., "Characterizing, modeling, and generating workload spikes for stateful services," in *SoCC'10*, (NY, USA), 2010.

[25] Cantrill, B., Shapiro, M. W., and Leventhal, A. H., "Dynamic instrumentation of production systems.," in *Proceedings of the 2004 USENIX Annual Technical Conference*, pp. 15–28, 2004.

[26] Cecchet, E., Marguerite, J., and Zwaenepole, W., "C-JDBC: Flexible database clustering middleware," in *Proceedings of the 2004 USENIX Annual Technical Conference, FREENIX Track*, pp. 9–18, 2004.

[27] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E., "Pinpoint: Problem determination in large, dynamic internet services," in *Proceedings of the 32th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2002)*, pp. 595–604, 2002.

[28] Chen, Y., Iyer, S., Liu, X., Milojicic, D., and Sahai, A., "Translating service level objectives to lower level policies for multi-tier services," *Cluster Computing*, 2008.

[29] Cho, S. and Jin, L., "Managing distributed, shared l2 caches through os-level page allocation," MICRO '06, 2006.

[30] Choi, K., Soma, R., and Pedram, M., "Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times," in *Proceedings of the 2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004)*, pp. 4–9, 2004.

[31] Cohen, I., Chase, J. S., Goldszmidt, M., Kelly, T., and Symons, J., "Correlating instrumentation data to system states: A building block for automated diagnosis and control.," in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, pp. 231–244, 2004.

[32] Dean, J. and Barroso, L. A., "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[33] Denning, P. J. and Buzen, J. P., "The operational analysis of queueing network models," *ACM Comput. Surv.*, vol. 10, no. 3, 1978.

[34] Denning, P. J. and Buzen, J. P., "The operational analysis of queueing network models," *ACM Comput. Surv.*, 1978.

[35] Diao, Y., Hellerstein, J., Storm, A., Surendra, M., Lightstone, S., Parekh, S., and Garcia-Arellano, C., "Using MIMO linear control for load balancing in computing systems," in *American Control Conference*, pp. 2045–2050, 2004.

[36] Erlingsson, Ú., Peinado, M., Peter, S., Budiu, M., and Mainar-Ruiz, G., "Fay: Extensible distributed tracing from kernels to clusters," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 4, p. 13, 2012.

[37] Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I., "X-trace: a pervasive network tracing framework," in *NSDI'07*.

[38] Forell, T., Milojicic, D., and Talwar, V., "Cloud management: Challenges and opportunities," in *IPDPSW'11*.

[39] Franks, G., Petriu, D., Woodside, M., Xu, J., and Tregunno, P., "Layered bottlenecks and their mitigation," in *QEST '06: Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems*, 2006.

[40] Freeh, V. W., Lowenthal, D. K., Pan, F., Kappiah, N., Springer, R., Rountree, B. L., and Femal, M. E., "Analyzing the energy-time trade-off in high-performance computing applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 6, pp. 835–848, 2007.

[41] Gettys, J. and Nichols, K., "Bufferbloat: dark buffers in the internet," *Communications of the ACM*, vol. 55, no. 1, pp. 57–65, 2012.

[42] Govindan, S., Liu, J., Kansal, A., and Sivasubramaniam, A., "Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines," in *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC 2011)*, p. 22, 2011.

[43] Gupta, V., Brett, P., Koufaty, D., Reddy, D., Hahn, S., Schwan, K., and Srinivasa, G., "The forgotten 'uncore': On the energy-efficiency of heterogeneous cores," in *Proceedings of the 2012 USENIX Annual Technical Conference*, 2012.

[44] Harchol-Balter, M., Schroeder, B., Bansal, N., and Agrawal, M., "Size-based scheduling to improve web performance," *ACM Trans. Comput. Syst.*, 2003.

[45] IETF, "RFC 6298." "http://tools.ietf.org/html/rfc6298".

[46] Isci, C., Contreras, G., and Martonosi, M., "Live, runtime phase monitoring and prediction on real systems with application to dynamic power management," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39)*, pp. 359–370, 2006.

[47] Kapoor, R., Porter, G., Tewari, M., Voelker, G. M., and Vahdat, A., "Chronos: Predictable low latency for data center applications," in *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC 2012)*, pp. 9:1–9:14, 2012.

[48] Kohavi, R. and Longbotham, R., "Online experiments: Lessons learned," *Computer*, vol. 40, no. 9, pp. 103–105, 2007.

[49] Li, J., Sharma, N. K., Ports, D. R., and Gribble, S. D., "Tales of the tail: Hardware, os, and application-level sources of tail latency," Tech. Rep. UW-CSE14-04-01, Department of Computer Science & Engineering, University of Washington, April 2014.

[50] Lim, H. C., Babu, S., and Chase, J. S., "Automated control for elastic storage," in *ICAC'10*, (NY, USA), 2010.

[51] Liu, X., Sha, L., Diao, Y., Froehlich, S., Hellerstein, J., and Parekh, S., "Online response time optimization of apache web server," *Quality of ServiceIWQoS 2003*, pp. 153–153, 2003.

[52] Malkowski, S., Hedwig, M., Parekh, J., and Pu, C., "Bottleneck detection using statistical intervention analysis," in *DSOM'07*, (Berlin, Heidelberg), 2007.

[53] Merkel, A., Stoess, J., and Bellosa, F., "Resource-conscious scheduling for energy efficiency on multicore processors," in *Proceedings of the 5th European conference on Computer systems (EuroSys 2010)*, pp. 153–166, 2010.

[54] Mi, N., Casale, G., Cherkasova, L., and Smirni, E., "Burstiness in multi-tier applications: Symptoms, causes, and new models," in *Proceedings of the ACM/IFIP/USENIX 9th International Middleware Conference (Middleware 2008)*, pp. 265–286, 2008.

[55] Mi, N., Casale, G., Cherkasova, L., and Smirni, E., "Injecting realistic burstiness to a traditional client-server benchmark," in *Proceedings of the 6th International Conference on Autonomic computing (ICAC 2009)*, pp. 149–158, 2009.

[56] Miftakhutdinov, R., Ebrahimi, E., and Patt, Y. N., "Predicting Performance Impact of DVFS for Realistic Memory Systems," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*, 2012.

[57] Min Lee, K. S., "Region scheduling: Efficiently using the cache architectures via page-level affinity," ASPLOS '12, 2012.

[58] Novaković, D., Vasić, N., Novaković, S., Kostić, D., and Bianchini, R., "DeepDive: Transparently identifying and managing performance interference in virtualized environments," in *Proceedings of the 2013 USENIX Annual Technical Conference*, pp. 219–230, 2013.

[59] Oliveira, F., Tjang, A., Bianchini, R., Martin, R. P., and Nguyen, T. D., "Barricade: defending systems against operator mistakes," EuroSys '10.

[60] Olshefski, D. and Nieh, J., "Understanding the management of client perceived response time," in *SIGMETRICS '06/Performance '06: Proceedings of the joint International conference on Measurement and modeling of computer systems*, (New York, NY, USA), pp. 240–251, 2006.

[61] Osogami, T. and Kato, S., "Optimizing system configurations quickly by guessing at the performance," in *Proceedings of the 2007 ACM SIGMETRICS International conference on Measurement and modeling of computer systems*, p. 156, 2007.

[62] Padala, P., Hou, K.-Y., Shin, K. G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., and Merchant, A., "Automated control of multiple virtualized resources," EuroSys '09.

[63] Pariag, D., Brecht, T., Harji, A., Buhr, P., Shukla, A., and Cheriton, D. R., "Comparing the performance of web server architectures," *In Proc. EuroSys '07*, 2007.

[64] Patterson, D. A., "Latency lags bandwith," *Communications of the ACM*, vol. 47, no. 10, pp. 71–75, 2004.

[65] Prasad, V., Cohen, W., Eigler, F., Hunt, M., Keniston, J., and Chen, B., "Locating system problems using dynamic instrumentation," in *Proceedings of the 2005 Ottawa Linux Symposium*, pp. 49–64, 2005.

[66] Raghavachari, M., Reimer, D., and Johnson, R., "The Deployer's Problem: Configuring Application Servers for Performance and Reliability," 2003.

[67] Reynolds, P., Killian, C. E., Wiener, J. L., Mogul, J. C., Shah, M. A., and Vahdat, A., "Pip: Detecting the unexpected in distributed systems.," in *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'06)*, pp. 115–128, 2006.

[68] Ruan, Y. and Pai, V. S., "Making the" box" transparent: System call performance as a first-class result.," in *Proceedings of the 2004 USENIX Annual Technical Conference*, pp. 1–14, 2004.

[69] Rumble, S. M., Ongaro, D., Stutsman, R., Rosenblum, M., and Ousterhout, J. K., "It's time for low latency," in *Proceedings of the 13th USENIX Workshop on Hot Topics in Operating Systems (HotOS 13)*, pp. 11–11, 2011.

[70] Sambasivan, R. R., Zheng, A. X., De Rosa, M., Krevat, E., Whitman, S., Stroucken, M., Wang, W., Xu, L., and Ganger, G. R., "Diagnosing performance changes by comparing request flows," in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, pp. 43–56, 2011.

[71] Schroeder, B., Wierman, A., and Harchol-Balter, M., "Open versus closed: a cautionary tale," NSDI'06, (Berkeley, CA, USA), USENIX Association, 2006.

[72] Sigelman, B., Barroso, L., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C., "Dapper, a large-scale distributed systems tracing infrastructure," in *Google Technical report'10*.

[73] Snowdon, D. C., Le Sueur, E., Petters, S. M., and Heiser, G., "Koala: A platform for OS-level power management," in *Proceedings of the 4th ACM European conference on Computer systems (EuroSys 2009)*, pp. 289–302, 2009.

[74] Snyder, B., "Server virtualization has stalled, despite the hype," *InfoWorld*, December 2010.

[75] Thereska, E. and Ganger, G. R., "Ironmodel: robust performance models in the wild," SIGMETRICS '08.

[76] Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., and Tantawi, A., "An analytical model for multi-tier internet services and its applications," in *SIGMETRICS'05*.

[77] Urgaonkar, B., Shenoy, P., Chandra, A., and Goyal, P., "Dynamic provisioning of multi-tier internet applications," in *Proceedings of the 2nd IEEE International Conference on Autonomic computing (ICAC 2005)*, pp. 217–228, 2005.

[78] Wang, Q., Kanemasa, Y., Kawaba, M., and Pu, C., "When average is not average: large response time fluctuations in n-tier systems," in *Proceedings of the 9th International Conference on Autonomic computing (ICAC 2012)*, pp. 33–42, 2012.

[79] Wang, Q., Kanemasa, Y., Li, Jack Lai, C.-A., Matsubara, M., and Pu, C., "Impact of dvfs on n-tier application performance," in *Proceedings of ACM Conference on Timely Results in Operating Systems (TRIOS 2013)*, pp. 33–42, 2013.

[80] Wang, Q., Kanemasa, Y., Li, J., Jayasinghe, D., Shimizu, T., Matsubara, M., Kawaba, M., and Pu, C., "Detecting transient bottlenecks in n-tier applications through fine-grained analysis," in *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems (ICDCS 2013)*, pp. 31–40, 2013.

[81] Wang, Q., Malkowski, S., Kanemasa, Y., Jayasinghe, D., Xiong, P., Pu, C., Kawaba, M., and Harada, L., "The impact of soft resource allocation on n-tier application scalability," in *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011)*, pp. 1034–1045, 2011.

[82] Welsh, M., Culler, D., and Brewer, E., "SEDA: An architecture for well-conditioned, scalable internet services," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, pp. 230–243, 2001.

[83] Xiong, P., Wang, Z., Malkowski, S., Wang, Q., Jayasinghe, D., and Pu, C., "Economical and robust provisioning of n-tier cloud workloads: A multi-level control approach," in *Proceedings of the 31st IEEE International Conference on Distributed Computing Systems (ICDCS 2011)*, pp. 571–580, 2011.

[84] Xu, Y., Musgrave, Z., Noble, B., and Bailey, M., "Bobtail: Avoiding long tails in the cloud," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, pp. 329–342, 2013.

[85] Zheng, W., Bianchini, R., and Nguyen, T., "Automatic configuration of internet services," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, p. 229, 2007.

[86] Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A., Padala, P., and Shin, K., "What does control theory bring to systems research?," *SIGOPS Oper. Syst. Rev.*, 2009.